12 (18) ARI TECHNICAL REPORT

(19) TR-78-A21

(6)

Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging.

(9) Technical rept. 12 Jul 56- 11 Nov 77,

by

(10) Michael E./Atwood ~~and~~ H. Rudy/Ramsey

SCIENCE APPLICATIONS, INC.
7935 East Printice Avenue
Englewood, Colorado 80110

(12) 90p.

(11) August 1978

(14) SAI-78-054-DEN

Contract DAHC19-76-C-0040

(15)

(16) 2Q762725A778

Monitored technically by
Edgar M. Johnson and Jean Hooper
Battlefield Information Systems Technical Area, ARI

Prepared for

ari

U.S. ARMY RESEARCH INSTITUTE
for the BEHAVIORAL and SOCIAL SCIENCES
5001 Eisenhower Avenue
Alexandria, Virginia 22333

392 878

U. S. ARMY RESEARCH INSTITUTE

FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the Deputy Chief of Staff for Personnel

JOSEPH ZEIDNER	WILLIAM L. HAUSER
Technical Director (Designate)	Colonel, US Army
	Commander

Rsearch accomplished
under contract to the Department of the Army

Science Applications, Inc.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-78-A21	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) COGNITIVE STRUCTURES IN THE COMPREHENSION AND MEMORY OF COMPUTER PROGRAMS: AN INVESTIGATION OF COMPUTER PROGRAM DEBUGGING		5. TYPE OF REPORT & PERIOD COVERED Technical Report 12 July 1976-11 November 1977
		6. PERFORMING ORG. REPORT NUMBER SAI-78-054-DEN
7. AUTHOR(s) Michael E. Atwood and H. Rudy Ramsey		8. CONTRACT OR GRANT NUMBER(s) DAHC19-76-C-0040
9. PERFORMING ORGANIZATION NAME AND ADDRESS Science Applications, Inc. 7935 E. Prentice Avenue Englewood, Colorado 80110		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q762725A778
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Institute for the Behavorial and Social Sciences (PERI-OS) 5001 Eisenhower Avenue, Arlington, Virginia 22333		12. REPORT DATE August 1978
		13. NUMBER OF PAGES 85
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Monitored technically by Edgar M. Johnson and Jean Hooper, Battlefield Information Systems Technical Area, ARI.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer programming, debugging (computers), information processing, memory (psychology), psychology.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A theoretical framework, based upon recent studies in cognitive psychology on memory for text, was developed to explain certain aspects of human behavior during computer program comprehension and debugging. A central concept of this framework is that the information contained in a program is represented in a programmer's memory as a connected, partially ordered list (hierarchy) of "propositions" (units of information with properties similar to those observed in research on text memory). An experiment was over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 10 31 004

performed to test the hypothesis that the difficulty in finding a program bug is a function of the bug's location in this hierarchy. This experiment compared the effects of bug location, bug type (array, iteration, assignment) and specific program. Each of 48 subjects debugged two separate programs, with one type of bug at two different hierarchical levels in each program.

A preliminary analysis suggested that all three factors -- program, bug type, and bug location -- significantly affected the time required to locate program bugs. Detailed analyses, however, suggested the program and bug type variables could be explained in terms of the bug location variable and that a bug's location in a program's underlying propositional hierarchy is a principal factor affecting performance in a comprehension and debugging task.

The Battlefield Information Systems Technical Area of the Army Research Institute is concerned with the human resource demands of increasingly complex battlefield systems used to acquire, transmit, process, disseminate, and utilize information. This increased complexity places great demands upon the operator interacting with the machine system. Research in this area is focused on human performance problems related to interactions within command and control centers as well as issues of system development. It is concerned with such areas as software development, topographic products and procedures, tactical symbology, user oriented systems, information management, staff operations and procedures, and sensor systems integration and utilization.

One area of special interest involves the development of computer software to support automated battlefield systems. Software development is a costly, unreliable, not well understood process. The present research developed a theoretical framework for the study of software production tasks based on the representation of text information in memory. An initial validation of this framework using tasks of program comprehension and debugging experimentally verified its potential power. The research is part of a larger effort to develop a conceptualization of the programming process and identify behavioral bottlenecks in software development. Efforts in this area are directed at improving accuracy and productivity in programming through the design of procedures, languages, and methods to enhance programmer performance.

Research in the area of human factors in software development is conducted as an in-house effort augmented contractually by organizations selected as having unique capabilities and facilities. The effort is responsive to requirements of Army Project 2Q762725A778, and to general requirements expressed by members of the Integrated Software Research and Development Working Group (ISRAD).

BRIEF

Requirement:

To develop and empirically test a theoretical framework capable of guiding and integrating future research on the psychological aspects of software development.

Procedure:

A theoretical framework, based upon recent studies in cognitive psychology on memory for text, was developed to explain certain aspects of human behavior during computer program comprehension and debugging. A central concept of this framework is that the information contained in a program is represented in a programmer's memory as a connected, partially ordered list (hierarchy) of "propositions" (units of information with properties similar to those observed in research on text memory). An experiment was performed to test the hypothesis that the difficulty in finding a program bug is a function of the bug's location in this hierarchy. This experiment, which was based on a paradigm previously used by others, compared the effects of bug location, bug type (array, iteration, assignment) and specific program. Subjects were 48 undergraduate students with a minimum experience level of three computer science courses. Each subject debugged two separate programs, with one type of bug at two different hierarchical levels in each program.

Findings:

A preliminary analysis suggested that all three factors -- program, bug type, and bug location -- significantly affected the time required to locate program bugs. Detailed analyses, however, suggested the program and bug type variables could be explained in terms of the bug location variable and that a bug's location in a program's underlying propositional hierarchy is a principal factor affecting performance in a comprehension and debugging task.

Utilization of Findings

The difficulty in detecting a bug is a function of the bug's location in the propositional hierarchy that a programmer constructs to represent a program. Programs should be written so as to avoid unnecessarily deep hierarchies and modularized so as to minimize the number of propositions in a given section in order to facilitate computer program comprehension and debugging. The principal contribution of this research is the demonstration of an initial theoretical framework which, with further development, may assist considerably in the integration of research findings and the guidance of future research on the psychological aspects of software development.

TABLE OF CONTENTS

TABLES

FIGURES

[illegible]

INTRODUCTION

The production of software is primarily a human activity. Although tremendous developments in both hardware and software technology have been made in the past decade, the analysis, design, coding, debugging, and testing and validation of software systems remains a predominantly human activity. Many of the software development methods and "rules of thumb" that have evolved for program design and implementation are intended to make software easier for humans to conceptualize, specify, communicate and proceduralize.

Research in this area, however, has been slow to materialize. A basic impediment to research in the psychological aspects of software development has been, and continues to be, the absence of a sound theoretical framework on which to build. Such a framework in the area of "software psychology," could make research efforts more productive in understanding how people write, comprehend, and debug computer programs, in developing better and more consistent programming practices, and in designing better programming languages and debugging aids.

This paper focuses on program comprehension and debugging and attempts to relate debugging performance to current psychological theories of comprehension and memory. The objective is to demonstrate a theoretical framework for studying software development tasks, in general, and program comprehension in particular, and to provide empirical support for this framework.

The principal empirical studies that have considered program comprehension and debugging are reported by Gould and Drongowski (1974) and Gould (1973). In these closely related studies, college student programmers were given FORTRAN programs that were syntactically correct, but contained a nonsyntactic error in one line. The subjects were asked to locate the line containing the error.

It is important to recognize that this task involves program comprehension as well as debugging strategy and behavior; a subject must comprehend a program before he can locate errors. Although this task is somewhat different from a debugging task in which a programmer must locate errors in a program he has written, it is very similar to the task of the maintenance programmer, who must understand and debug or modify someone else's program. Unlike the task of debugging one's own program, this task allows for appropriate experimental manipulations and control.

Each of these experiments used four statistical programs from the IBM Scientific Subroutine package. Three versions of each program were prepared by introducing into each program one of three different types of bugs. "Assignment" bugs changed a variable in an assignment statement, "array" bugs caused an array to exceed its dimensions, and "iteration" bugs caused an incorrect number of iterations through a loop.

In both studies, each programmer attempted to find the error in each of the twelve programs. Performance was generally good, with median debug times of 6-7 minutes. A strong learning effect was evident, with delay times roughly cut in half when the same program was debugged (with a different bug) the second time. This is consistent with the view that this task has a strong comprehension component. Perhaps the most interesting finding observed in both studies was that "assignment" bugs were much more difficult to find than either "array" or "iteration" bugs, which were roughly equal in difficulty.

There are several possible explanations for this result. First, as the authors point out, array and iteration bugs may be detectable on the basis of a less detailed understanding of the program than is required to detect assignment bugs. Subjects who adopted a strategy of comparing loop indices with dimensioned array sizes might be able to detect array and iteration bugs without really understanding the functions of the program. Second, the assignment bugs (especially the more difficult ones) tended to involve errors in statistical formulas and, therefore, more non-program-related knowledge was required for their detection than was required for array or iteration bugs. This is, of course, more a measure of statistical knowledge than of debugging ability and it is possible that assignment bugs that did not involve knowledge of statistics might be no more difficult to detect than other types of bugs. A third explanation, also suggested by Gould and Drongowski, relates debugging difficulty to the distance between the actual location of a bug and the location at which the effect of the bug can be detected. It should be noted that the same programs and bugs were used in both studies, so that any factors that may have been confounded with bug type are equally confounded in the two experiments.

A fourth explanation is suggested in this paper. Recent research in psychology has demonstrated that performance in a text comprehension task can be explained in terms of the manner in which textual information is represented in memory. That is, performance is strongly influenced by the logical structure of the representation that a subject forms during text comprehension. Programs appear to have similar logical structures that have similar effects on program comprehension and debugging.

The next section presents a brief review of the theories and issues that have been developed by cognitive psychologists in studies of the representation of knowledge. The following two sections use these theories to develop a framework for describing the representation of computer programs in memory and describe some preliminary analyses conducted to evaluate this framework. Finally, the results of an initial study of computer program debugging are described in terms of this theoretical framework.

THEORETICAL CONSIDERATIONS

The study of cognitive psychology can roughly be divided into two theoretical areas -- the representation of knowledge and the utilization of knowledge. These areas are generally referred to as "memory" and "problem solving," respectively. The representation issue is primarily concerned with how information is represented in memory; the utilization issue centers on the skills and processes that are used to utilize this information. Any complete theory of programming must eventually consider both the issue of knowledge representation and that of knowledge utilization. In our initial studies, however, it appears more productive to focus on the representation issue.

This section reviews some of the theories and issues in studies of the representation of knowledge. The primary purpose of this section is to introduce the terminology and paradigms used in this area and to present typical research results. The following section demonstrates how these theories and paradigms can be applied to the study of programming.

In developing a theory for the representation of programs, we will follow closely the theory of text memory proposed by Kintsch (1974). In Kintsch's (1976) formulation, "the meaning of <u>text</u> is represented by its <u>text base</u>. The text base consists of a connected, partially ordered list of <u>propositions</u>. Propositions contain one or more <u>arguments</u> plus one <u>relational term</u>." Consider first the concept of a proposition. Basically, a proposition makes a meaningful statement. It is convenient to think of a proposition as the least amount of textual material that can convey an idea. For example, "the dog" would not be a meaningful proposition, but "the dog barks" would be.

A text base consists of a partially ordered <u>hierarchy</u> of propositions. That is, one or more propositions will represent the superordinate, or most important, ideas to be expressed and other propositions will give more information about these ideas.

To illustrate these concepts, consider the following two sentences (from Kintsch, 1974).

(1) Romulus, the legendary founder of Rome, took the women of the Sabine by force.

(2) Cleopatra's downfall lay in her foolish trust in the fickle political figures of the Roman world.

The suggested propositions underlying the first sentence are:

1. (TOOK, ROMULUS, WOMEN, BY FORCE)
2. (FOUND, ROMULUS, ROME)
3. (LEGENDARY, ROMULUS)
4. (SABINE, WOMEN)

The first term in each proposition is a relational term (predicate) and the remaining terms are arguments. These propositions express the ideas that:

1. Romulus took the women by force.
2. Romulus founded Rome.
3. Romulus is legendary.
4. The women are Sabine.

The second sentence is composed of the propositions:

1. (BECAUSE, α, β)
2. (FELL DOWN, CLEOPATRA) = α
3. (TRUST, CLEOPATRA, FIGURES) = β
4. (FOOLISH, TRUST)
5. (FICKLE, FIGURES)
6. (POLITICAL, FIGURES)
7. (PART OF, FIGURES, WORLD)
8. (ROMAN, WORLD)

The hierarchies of these sentences are:

(1)
$$\begin{tikzcd}
 & 2 \\
1 \arrow[ur] \arrow[r] \arrow[dr] & 3 \\
 & 4
\end{tikzcd}$$

(2)
$$\begin{tikzcd}
 & 2 & & \\
1 \arrow[ur] \arrow[r] & 3 \arrow[r] \arrow[dr] \arrow[ddr] & 4 & \\
 & & 5 \arrow[r] & 6 \\
 & & 7 \arrow[r] & 8
\end{tikzcd}$$

Consider now the type of results that have emerged in research in test memory as a result of adopting such a framework. First, propositions, rather than individual words, appear to be the basic unit of recall. For example, Kintsch and Keenan (1973) demonstrated that the amount of time a subject takes to read a text passage is positively related to the number of propositions presented, and that the number of propositions that can be recalled is also positively related to reading time. A propositional analysis also leads to predictions about which propositions will be most readily recalled. A typical result is that the probability of recall is inversely related to the proposition's level in the hierarchy. That is, the higher a proposition is in the hierarchy, the more likely it is to be recalled.

Consider again the two sentences given above as examples. Although they contain approximately the same number of words, the second contains twice as many propositions as the first. This implies that the second sentence will take longer to read than the first, that the proportion of propositions correctly recalled will be lower for the second sentence than for the first, and that the propositions in the second sentence least likely to be recalled are propositions number 6 and 8. (cf. Kintsch, 1974).

One additional result deserves mention because we will attempt to apply it directly to programming in a later section. Consider the following stories (Kintsch, 1974).

(3) Police are hunting a man in hiding. The man is Bob Birch, whose wife disclosed illegal business practices in an interview on Saturday.

(4) Police are hunting a man in hiding. The wife of Bob Birch disclosed illegal business practices in an interview on Saturday.

Story (3) explicitly expresses information that is implicit in story (4); namely, that "Bob Birch is the man who is hiding." A typical result is that stories such as (3) take longer to read; that is consistent with the results noted above, since the explicit information increases the number of propositions in the text. The principal result, however, is that even though subjects are very accurate at verifying the truth of test sentences, reaction time for verifying implicit sentences is much longer than those for explicit sentences.

Consider again story (4) above and how we can infer the implicit information. We have a fair amount of knowledge about police, and we know under what conditions the police would be looking for a man who is hiding. Specifically, this pre-existing knowledge contains the information that the man in question has probably done something illegal; this allows us to make the connection with "Bob Birch" since the story tells us that he has probably broken a law. This information about police is part of a "police schema," where a schema is a data structure for representing and integrating related bits of information.

When we read a story, we try to find some way to integrate and represent the presented information. In this case, Kintsch and van Dijk (1975) suggest that a schema functions like an outline with empty slots.

That is, a schema is a set of expectations about what the story will consist of. When these expectations (or slots) are satisfied, this outline becomes the macro-structure of the text. That is, a schema tells one, in general, what to expect and a macrostructure describes how these expectations are satisfied. Macrostructure propositions are, in essence, the superordinate propositions in the hierarchy and serve primarily to relate the remaining propositions.

While a macrostructure describes the semantic aspects of a story, a story grammar describes the syntactic aspects. For example, just as a sentence can be decomposed into a noun phrase and a verb phrase, a prototypical narrative involves a setting, a protagonist, some complication, and a resolution of that complication. Just as it is difficult to read an ungrammatical sentence, it is difficult to read an ungrammatical story. This is essentially the explanation offered by Mandler and Johnson (1977) in describing some of Bartlett's (1932) results. (The story in question is an American Indian tale, and its grammatical structure is obviously different, even to the casual reader, than that of Indo-European tales). As we will point out later, story grammars are similar to some of the proposed formal definitions of programming languages.

These results are not exhaustive of those obtained by a propositional analysis of text, but are representative of the types of analyses that can be performed.

THE REPRESENTATION OF PROGRAMS

This analysis focuses primarily on the cognitive structures that are developed during program comprehension. That is, the analysis concentrates on how information about programs is represented in memory rather than on the skills and processes that are used to utilize this information. Although it is impossible to separate cognitive structures from problem solving processes, since these structures are the results of such processes, it is not necessary (or, perhaps, currently practical) to consider both simultaneously, in developing a first-order theory of performance.

There appears to be no task, which has been studied in text memory research, that is a reasonable analog to program debugging. The type of theoretical framework discussed above, however, allows us to make some reasonable conjectures about the nature of debugging. The framework outlined below is only loosely defined, although it is consistent with theoretical developments in text memory research. The applicability of these ideas to program comprehension is demonstrated in the following two sections.

Consider first the definition of a bug. Programming errors can be divided into two broad classes -- syntactic and non-syntactic. Syntactic errors are detected by compilers and assemblers and are not of interest here. Non-syntactic errors (or "bugs"), however, present a more interesting problem. Such bugs represent (or are embedded in) legal program statements. A "bug", therefore, does not exist in isolation; rather a bug only exists within the context of a given program. Although this is an obvious statement, it does, when combined with the framework described above, produce a useful description of program debugging.

When we read text, we try to find some way to integrate the presented information into a coherent propositional hierarchy. If we are not able to do this, we generally label the text as "unconnected" or "nonsense". In some cases, as Paige and Simon (1966) illustrated in their study of algebra word problems, we may even misperceive the presented information so that a coherent representation can be formed. Forming a coherent propositional representation is essential for the comprehension of text.

Constructing a coherent representation also appears to be essential for program comprehension. When a programmer reads a program, he processes a relatively small number of propositions at a given time. He may process a single line, a group of lines, or only part of a single line; the maximum length of the program segment processed is determined by the number of propositions, or amount of information, that the programmer can readily assimilate rather than by a fixed number of lines.

As each segment is read, the actual code, and supporting comments, are decomposed into the underlying propositions and these propositions are integrated into the hierarchy that is being constructed to represent the entire program. The presence of a bug is noted when this integration fails. A bug, in general, contains information that contradicts the information contained in other propositions.

For the sake of simplicity, we are assuming that the program statements are accurately decomposed into the underlying propositions. We are not considering those cases in which the statement or proposition containing the bug is overlooked or misinterpreted. In these cases, it may be possible to successfully integrate the perceived propositions into a hierarchy, but this hierarchy would not accurately reflect the given program. Such a case is regarded as a "special case" and is beyond the scope of the present study. (For the interested reader, however, we will return to this problem briefly in our discussion section where we note a negative correlation between experience and debugging performance.)

Although failure to integrate propositions into a coherent hierarchy is assumed to indicate the presence of a bug, it does not necessarily unambiguously indicate the location or nature of the bug. Correcting a bug involves locating the proposition, or propositions, that must be changed in order to form a coherent hierarchy and then determining the appropriate changes.

Locating a bug in a previously unseen program involves forming a hypothesis about the nature of the propositional hierarchy required to adequately represent the program and then testing this hypothesis. Stated in other terms, locating a bug requires that hypotheses be formed about the functions that individual program segments are to perform and the interrelations between these functions. This type of behavior is most readily apparent when we observe programmers using "test case" input data, causing the partial results of calculations to be displayed, etc.

In order to investigate knowledge representation issues in program comprehension, it is necessary to describe the propositions underlying the programming language to be used, which is FORTRAN in this study, and the manner in which these propositions are related to form a hierarchical representation of a program. In the absence of research that indicates the form of such propositions, any proposed mapping of programs into propositional structures will be speculative and somewhat subjective. The validity of a mapping can only be determined by measuring its ability to account for empirically observed results.

An acceptable mapping of programs into propositional structures must not only adequately represent the various program statements, but must also allow for individual differences in representing programs in memory and be compatible with theories of related software development tasks and theories of the problem solving processes that utilize these structures.

The topic of individual differences is related to that of problem solving processes. A large variation in programmer performance has been frequently observed (e.g., Grant and Sackman, 1967). Some of these differences are primarily due to the fact that some programmers have developed more effective programming skills, or problem solving processes, than others and some differences are due primarily to differences in the manner in which programs are represented in memory. Even in this latter case, however, it is difficult to exclude the effects of problem solving processes. The principal factor influencing program representations is experience and experience is closely related to problem solving processes. Although the effects of experience will be discussed in more detail below, a brief example will serve to illustrate how individual differences may affect the propositional representation that is constructed to represent a given program.

Miller and Becker (1974) have demonstrated that nonprogrammers tend to express algorithms in a procedurally non-explicit form. Although this research was conducted with nonprogrammers with program-like tasks, it illustrates an important point that can be generalized both to programmers and to non-program related tasks. There is no need to explicitly represent detailed procedural information if this information can easily be reconstructed when it is needed. That is, if a subject has sufficient experience with a given task domain, he does not need to form a detailed representation of that task in memory; rather, he can construct a much more efficient representation and apply problem solving processes to that representation whenever it is necessary to reconstruct the procedural detail.

For example, it is likely that experienced programmers encode a program segment such as

```
      SUM = 0.
      DO 1 I = 1, N
         SUM = SUM + X(I)
    1 CONTINUE
```

as "CALCULATE THE SUM OF ARRAY X" rather than in some more procedurally explicit form. Problem solving processes could then allow a programmer to write code that is functionally equivalent to, although perhaps not identical to, this program segment.

The existence of individual differences must be considered in determining the propositions that underlie a programming language. To be useful, a theoretical framework based on propositions must also be compatible with the development of similar propositional structures during the software design process (cf. Brooks, 1975) and related software development tasks.

In order to reduce the subjective nature of determining propositional structures, propositions that are closely related to the form of the statements in the programming language are used in this study. Thus, we have postulated a "DO" proposition (for DO-loops) which has segments that correspond to the variables employed in a DO-loop, rather than adopting some other form of representation. This was done in an effort to make the mapping from programming language into propositional form as well-specified as possible. The structural relationship among propositions, which is of principal importance in the present study, is assumed to be relatively unaffected by assumptions concerning the form of the individual propositions. A proposed description of the propositional form of some of the statements used in the current study is shown in Figure 1.

For assignment statements, we have selected the predicate SET and two arguments -- VARIABLE and VALUE. The VALUE argument can take either of two forms. It can represent a numeric value, in which case we represent it directly as in SET(I,10); or it can take the form of a variable which we will represent as an embedded proposition as in SET(I, VALUE(J)). In this case, VALUE(J) is a subordinate proposition to the SET proposition.

In representing logical and arithmetic IF's, we use one higher-order proposition, LIF or AIF respectively, to indicate the type of IF statement. This convention was adopted since both types of statement are referred to in FORTRAN by the same name -- IF. The lower order propositions describe the specific conditions specified in the IF statement.

Statement Type	Propositional Form	Example	

Assignment	SET [VARIABLE, VALUE]	I = 10
		I = J
			SET (I,10)
			SET (I,VALUE(J))

DO Loop	DO [VARIABLE, FROM-VALUE, TO-VALUE, SCOPE]	DO 1 I=1,10 1 X(I)=0.	DO(I,1,10,a)——a=SET(X(I),0.)

Arithmetic IF	AIF [COND1,GOTO1, COND2, GOTO2, COND3, GOTO3]	IF(X) 1,2,3	AIF —— COND(X.LT.0)——GOTO(1)
			COND(X.EQ.0)——GOTO(2)
			COND(X.GT.0)——GOTO(3)

Logical IF	LIF [COND, ACTION, ALT-ACTION]	IF(X.EQ.0)X=1	LIF —— COND(X.EQ.0)——SET(X,1 GOTO(Next Statement)
GOTO	GOTO [LOC]	GOTO 10	GOTO(10)
Computed GOTO	CGOTO [(COND, GOTO)]	GOTO(1,2),I	CGOTO —— COND(I.EQ.1)——GOTO(1) COND(I.EQ.2)——GOTO(2)
Function CALL	FCALL [NAME, (ARG)]	SQRT(X)	FCALL(SQRT,X)
Subroutine CALL	SCALL [NAME, (ARG)]	CALL INIT(I,J)	SCALL(INIT,I,J)

Figure 1. Propositional Descriptions of Selected FORTRAN Statements

Notice that in the propositional representation of the logical IF (LIF), we have included an argument called ALT-ACTION (alternative action), and this argument is represented in the example by the proposition GOTO (next statement). This is an example of an "implicit proposition"; this information is not explicitly represented in a logical IF statement but it is essential for understanding this type of statement. Although we include this as an example of an implicit proposition, experienced programmers do not specifically attend to this proposition, just as they do not attend to the fact that each non-transfer-of-control statement has an implicit GOTO (next statement) proposition. This type of information is probably not, however, as easily comprehended by a novice programmer.

We have also omitted a potential argument (STEP-SIZE) from the DO proposition. This was done because this variable was not employed in the programs used in this study. There is, of course, an implicit proposition that this value is equal to one.

In representing the propositional structure of a program, we will introduce a proposition that employs the predicate SAME. This proposition does not correspond to a FORTRAN statement, but is used to describe the fact that certain propositions assign values to variables that are used elsewhere. Assume, for example, that the first proposition of a program is SET(N,30) and that a later proposition is SET(M,VALUE(N*N)). In this case, the value assigned to the variable M is a function of the value assigned earlier to N and we express this relation by making the proposition SAME(N,1) subordinate to this later proposition, where "1" identifies the first proposition. In effect, this proposition states that the current value of the variable N was set in the first proposition.

Propositions are related in a hierarchical manner; the resulting structure can be called the program (or text) base. In text, connections between propositions are determined by the repetition of arguments. One or more propositions will be the superordinate propositions in a text base, while any propositions sharing an argument with these propositions will form the second level of the hierarchy, and propositions having an argument in common with second level propositions, but not with first level propositions, will form the third level, etc. Argument repetition is necessary for continuity in text.

In determining the hierarchical representation of the propositions underlying a program, we will also employ an argument repetition rule. It is necessary, however, to define an additional rule. In a programming language, unlike text, one statement (or proposition) can exert "control" over other statements. For example, a conditional statement (e.g., logical IF) determines whether or not certain statements will even be executed (e.g., IF(X.EQ.0)I=I+1). Likewise, a DO statement exerts control over all statements in the body of the DO-loop, and the propositions underlying these statements will be at a lower level, or levels, than the DO proposition. In neither case, however, is argument repetition necessary. In these cases, it is "control" rather than argument repetition that determines a proposition's level in the program base hierarchy.

Thus far, we have assumed that a given program statement has a single, consistent underlying propositional structure. That is, in general, not true. An important factor that determines the type of representation a programmer will form for a given program is his level of experience. That is, more experienced programmers will form more efficient representations.

Support for this statement can be found in an experiment reported by Shneiderman (1976), which was modeled after a paradigm used by Chase and Simon (1973). Shneiderman compared the performance of experienced and inexperienced programmers on recalling programs in which the statement order was either normal (compiler-acceptable) or random. Experienced programmers were able to recall considerably more program statements than novice programmers on normal order programs, but there were no differences on scrambled order programs.

The superior recall of experienced programmers is due to their ability to form chunks; that is, they use fewer propositions or simpler propositions to represent a program segment than do less experienced programmers. The relationship between chunks and propositions will be considered in more detail in the following section, where chunks will be related to higher-order, or "macrostructure," propositions. In determining the form of a program's underlying propositional representation, therefore, it is necessary to consider the experience level of the programmers to be studied and the types of program chunks they can reasonably be expected to form.

The propositional representations described in Figure 1 correspond to the types of structures we would expect to be constructed by fairly inexperienced programmers. As they gain more experience, however, other propositions may be added to this list or this list might be otherwise modified. In order to illustrate this concept, we will consider two such cases that quite probably occur even with only moderately experienced programmers.

Statements of the general form "I=I+1" or "X=X+A(I)" are fairly common in FORTRAN programs. Although these assignment statements could be represented by SET(VARIABLE,VALUE) type propositions, familiarity with such statements probably leads to the development of a more efficient representation. For example, instead of representing I=I+1 as SET(I, VALUE(I+1)), a more experienced programmer may use the representation INCREMENT(I,1), or simply, INCREMENT(I) where the value "1" is assumed by default.

Similarly, consider the program segment:

```
DO 1 I = 1, 10
   A(I) = 0.0
   B(I) = 0.0
 1 C(I) = 1.0
```

Although this could be represented with DO and SET propositions, more experienced programmers probably form a propositional hierarchy like the following:

```
1. INITIALIZE(VECTORS)
2. SIZE(VECTORS,10)
3. SET(A,0.0)
4. SET(B,0.0)
5. SET(C,1.0)
```

$$\begin{tikzcd} & 2 \\ 1 \arrow[ur] \arrow[r] \arrow[dr] \arrow[ddr] & 3 \\ & 4 \\ & 5 \end{tikzcd}$$

Such a representation can be used whenever a programmer becomes familiar with variable initialization procedures and learns to recognize such procedures fairly quickly.

In both of these examples, the representations that could be formed by even moderately experienced programmers involve fewer propositions or propositions with fewer arguments than the representations that would be formed by less experienced programmers. In the programs to be described below, we have used these constructs since they appear to be compatible with the experience level of our subjects. These additional constructs, however, also closely follow the form of the statements used in FORTRAN.

The framework outlined above is, admittedly, only loosely defined. We have attempted to abstract the major ideas that were successfully applied in text memory research. The applicability of these ideas to program comprehension has not yet been demonstrated. In the following two sections, we will attempt to provide such a demonstration.

A PRELIMINARY TEST OF THE PROPOSITIONAL STRUCTURE OF PROGRAMS: ANALYSES OF EMPIRICAL EVIDENCE FROM EARLIER STUDIES

THE GOULD AND DRONGOWSKI DEBUGGING STUDY

To test the feasibility of a propositional representation of programs, the programs used by Gould and Drongowski (1974) were re-analyzed. One of the typical results in studies of text memory is that the probability of recalling a proposition is a function of that proposition's level in the hierarchy. The difficulty in locating a programming error may, likewise, be a function of the level of the proposition in which the error is embedded. There does not seem to be a reasonable analog of debugging in text memory. The closest counterpart may be ambiguous stories in which certain propositions may reasonably have more than one interpretation. For example, in a prototypical suspense story, we are deliberately confused or led to misperceive the "true" meaning of events or characters; the conclusion of the story forces us to re-interpret these meanings.

The task of debugging is roughly analogous to reading a suspense story. Except for obvious syntactic errors, a bug is represented by a proposition that is potentially true. That is, a programming error exists only in the context of the entire program or program segment. When we read a program, we accept all of the underlying propositions as true and attempt to integrate them into a meaningful hierarchy. When the program does not perform as expected, we assume that one or more propositions (embedded in one or more statements) are incorrect. Assume that such a statement has been located. Our initial reading and study of the program resulted in the formation of a coherent hierarchy. Changing this proposition may also necessitate changing other propositions in order to define a more accurate hierarchy. This leads to the hypothesis that the difficulty involved in correcting a bug is a function of the number of propositions that must be changed in order to implement the correction.

If this hypothesis is correct, two factors influence debugging time. First, the time to locate an error is a function of the error's level in the hierarchy. Second, the time to actually correct the error is a function of the number of propositions that must be changed. In practice, these two factors are confounded, since potential changes will be considered while the error is being sought. To illustrate these concepts, consider a portion of one of the programs used by Gould and Drongowski and its underlying propositional representation. The program segment is:

```
C      CALCULATE THE NUMBER OF CATEGORY INTERVALS
          N = (UBO(3)-UBO(1)/UBO(2) + 0.5
C      CALCULATE TOTAL FREQUENCY
          T = 0.0
          DO 110 I = 1,N
  110     T = T + F(I)
  115     NOP = 5
  120     JUMP = 1
```

We represent the propositional structure of this program as:

1. SET(N,VALUE((UBO(3)-UBO(1))/UBO(2) + 0.5)
2. SET(T,0.0)
3. DO(I,1,N,α)
4. SAME(N,1)
5. INCREMENT(T,F(I))
6. SAME(T,2)
7. SET(NOP,5)
8. SET(JUMP,1)

I. CALCULATE NUMBER OF CATEGORY INTERVALS
II. CALCULATE TOTAL FREQUENCY

$$\begin{tikzcd}
\mathrm{I} \arrow[r, dash] & 1 & & & \\
 & 2 & & & \\
\mathrm{II} \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dash] \arrow[ddr, dash] & 3 \arrow[r, dash] \arrow[dr, dash] & 4 & & \\
 & 7 & \alpha \arrow[r, dash] & 5 \arrow[r, dash] & 6 \\
 & 8 & & &
\end{tikzcd}$$

The two higher-order propositions "CALCULATE NUMBER OF CATEGORY INTERVALS" and "CALCULATE TOTAL FREQUENCY" represent part of the macrostructure of this program. These propositions are not explicitly represented in the actual code, but rather, describe the purpose and actions of the code, or subordinate propositions. In the present case, these macropropositions are represented by comments.

The propositional representation of this program segment is tentative. Although such a representation may exist in a programmer's memory, the exact form of this representation cannot be determined without conducting empirical studies. For example, we represented "T = T + F(I)" as "INCREMENT (T, F (I))". Alternative representations could be "SET (T, VALUE (T) + VALUE (F(I))" or "SET (T,B)" and "B = (VALUE (T) + VALUE (F(I)))", etc. Recall and response time studies should help to resolve this issue. In the absence of firm evidence, we adopted a reasonable representation as a first-order approximation that we could employ consistently. In addition, note that we did not explicitly include statement numbers in our representation of "DO LOOPS". This reflects our early assumption that statement numbers are not represented explicitly and their use in DO constructs is primarily to determine the appropriate level in the hierarchy for a given statement. Pilot studies to be reported below, have substantiated the latter assumption, and have also suggested that such factors as statement numbers are considered during the encoding process, but are not encoded in memory in a normal comprehension task.

The hypothesis about debugging time implies that an error in the statement "T = T + F (I)" would be more difficult to locate than an error in the statement "NOP = 5", since they appear at different levels in the hierarchy. Both "NOP = 5" and "JUMP = 1" would be equally difficult to locate, since they are at the same level. Consider now the task of correcting an error. The bug introduced by Gould and Drongowski was to change "T = 0" to "T = N". In our formulation, this introduces two extraneous propositions. Rather than "SET (T,0.)", the representation becomes "SET (T, VALUE (N))" and "SAME (N,1)". Correcting this bug, therefore, should be more difficult than if this statement were changed to "T = 1", which does not introduce additional propositions.

Data reported by Gould and Drongowski allow a preliminary test of this hypothesis. The propositional structure of three of their programs (MOME, TALL, and CORR) was determined. These three programs represent extreme values in program length and median time to debug. Assuming that debugging difficulty is a function of the number of propositions that must be changed times the level in the hierarchy where the error occurs, we obtained a correlation of approximately .94 between this measure and median time to debug. While suggestive, this is, of course, not conclusive, since it is not a test of an *a priori* hypothesis.

THE PROPOSITIONAL MACROSTRUCTURE

This analysis excluded one of the bugs introduced by Gould and Drongowski. This bug involved an incorrect representation of the formula for a standard deviation. It is not surprising that such a bug is difficult to correct. In reading text, the reader develops a macrostructure to aid in comprehending the text. This macrostructure contains, in part, real-world knowledge, which though not explicitly presented in the text, is necessary in order to comprehend the meaning of the text.

Consider the example, "I went to three drugstores" (Norman, 1973). When presented with such a statement, we would not passively encode it; we would probably ask the speaker "didn't the first two drugstores have what you wanted?" The macrostructure that we invoke to represent this statement involves knowledge of the motives and purposes for going to drugstores, and we use this knowledge in order to understand the statement. That is, we try to integrate new information with past experiences and knowledge. This attempted integration serves to insure that this new information is reasonable and consistent with what we already know.

In the case under consideration, an error was introduced that involved an incorrect statistical formula. Correcting this type of error would be greatly facilitated if subjects could readily recall the correct formula. The relatively long observed debugging times indicate that this is not the case. The experiences and pre-existing knowledge that are brought to the debugging task are a primary determinant of performance. (This suggests that individual differences in programming performance may be explainable, in part, in terms of factors that influence the formation of propositional program representations by individual programmers.)

The existence of macrostructures in text memory is easily demonstrated. Macrostructures function as prototypical outlines for different types of text. We have, for example, one macrostructure for journal articles, one for fairy tales, one for suspense stories, etc. With text, we determine the appropriate macrostructure fairly quickly and easily and proceed to build upon it.

When we consider programs, however, it is not so clear what types of macrostructures are involved. Shneiderman's (1976) results suggest that a "chunking" phenomenon is evident when recalling programs. While not a surprising result, it does indicate how a program macrostructure could aid program comprehension.

Consider the following program segment (from Shneiderman):

```
    DO 10 I = 1,40
    READ (5, 82) X
    IF (X.GT. TEST) GO TO 5
    XSMALL = XSMALL + X
    NSMALL = NSMALL + 1
    GO TO 10
  5 CONTINUE
    . . .
 10 CONTINUE
```

While the underlying propositional structure of this segment can be represented using the rules in Figure 1, the experienced programmer probably uses a more efficient representation. He encodes this segment as "read 40 values and count and sum those less than some criterion value." This representation contains fewer propositions than the actual program code, and is hence better remembered. In addition, notice that this alternative representation describes the purpose and effects of this program segment. It is, in effect, a higher-order, or macrostructure, proposition. The experienced programmer's superior performance in this task is probably due to his requirement of storing and recalling only the higher-order propositions, and his ability to produce the actual code by applying his knowledge of programming syntax and semantics to these propositions.

While this indicates that a macrostructure is developed as a program is studied, it does not indicate whether a programmer selects an appropriate macrostructure from a limited class of prototypic structures or whether each program produces a unique macrostructure description. On the one hand, the lack of such prototypic structures could explain why learning to write and comprehend programming languages is often considered more difficult than learning to write and comprehend natural language for which well-defined prototypes exist. On the other hand, the discovery of a class of program macrostructures could greatly increase our knowledge of programming behavior.

RECALL OF COMPUTER PROGRAMS

Studies of both text comprehension and program comprehension indicate that information about text and programs is stored in memory in a predominantly semantic form. That is, the information is stored in terms of its meaning relative to the subject's prior store of information, rather than in terms of the literal syntactic form in which the new information was received. A specific example may serve to illustrate this point and to convey a better idea of the form that propositions take in computer program comprehension.

Consider the program shown in Figure 2. This program was used (with different comments) in a not-yet-published study by Shneiderman (1977). We have also used it in several informal exploratory studies. One of these procedures requires the subject to read the program for comprehension (as opposed to memorizing it) and then, with the program removed from view, to reproduce the program from memory. The subject is told to exactly reproduce everything he can, and to reproduce everything else as closely as possible. Figure 3 is a transcription of one subject's handwritten reproduction of the program. With the exception of input/output formats and one omitted statement (also omitted on a delayed reproduction 5 days later), the reproduced program is functionally equivalent to the original. Yet it differs in many syntactic aspects.

Consider, for example, the program segment

```
     DO 10 I = 1,N
     READ(5,101)G(I)
101  FORMAT(10X,F8.1)
 10  CONTINUE
```

which was reproduced as

```
     READ(5,2)(G(I),I=1,N)
2    FORMAT(F10.2)
```

This and similar observations suggest that such program segments are not necessarily encoded on the basis of even such fairly dominant syntactic constructions as DO-groups, but are chunked into more functional units. This particular observation suggests that there is a "READ ARRAY" proposition and that, at least for experienced programmer subjects, little additional detail is explicitly stored in memory. Rather, the detail is reconstructed in the reproduction task according to mapping rules which have been learned by the subject. This is consistent with the results of text comprehension studies, and with studies of the recall of programs (e.g., Shneiderman & Mayer, 1975). It is of interest here, however, because it appears that

```
C
C ****************************************************************************
C
C      THE ARRAY G IS AN ARRAY OF STUDENTS' GRADES ON AN EXAMINATION.
C
C ****************************************************************************
C
      DIMENSION G(500)
      READ(5,100)N
  100 FORMAT(I3)
      DO 10 I=1,N
      READ(5,101)G(I)
  101 FORMAT(10X,F8.1)
   10 CONTINUE
      S=0.0
      DO 20 I=1,N
      S=S+G(I)
   20 CONTINUE
      A=S/FLOAT(N)
      J=1
      GJ=G(1)
      DEVGJ=ABS(GJ-A)
      DO 30 I=2,N
      DEV=ABS(G(I)-A)
      IF(DEVGJ.LE.DEV)GO TO 30
      DEVGJ=DEV
      J=I
      GJ=G(J)
   30 CONTINUE
      WRITE(6,102)J,GJ
  102 FORMAT('  ',I3,10X,F8.1)
      STOP
      END
```

Figure 2. A FORTRAN Program Used in Some Recall Studies

```
C ******************************************
C
C     G IS AN ARRAY OF STUDENTS' GRADES
C
C ******************************************
          DIMENSION G(500)
          READ(5,1) N
1         FORMAT(I3)
          READ(5,2) (G(I),I=1,N)
2         FORMAT(F10.2)
          S=0.
          DO 10 I=1,N
          S=G(I)+S
10        CONTINUE
          A=S/FLOAT(N)
          J=1
          GJ=G(J)
          DEVGJ=ABS(A-G(1))
          DO 100 I=2,N
          DEV=ABS(A-G(I))
          IF(DEVGJ.LE.DEV) GO TO 100
          J=I
          GJ=G(J)
100       CONTINUE
          WRITE(6,200) J,GJ
200       FORMAT('...',I3,F10.2)
          STOP
          END
```

Figure 3. Transcription of a Subject's Recollection of The Program of Figure 2.

such experimental procedures as these can help us identify the specific nature of the encoded propositions (in this case, perhaps "READ ARRAY", rather than "DO" with an underlying propositional substructure).

PROGRAMMING STYLE AND PRACTICES

Programmers, primarily through extensive trial-and-error experiences, have developed general principles, or "caveats" of "good programming" practices. If the thesis of this paper is correct, then "good" practices should differ from "bad" practices in their underlying propositional structure. As a further test of the propositional structure of programs, some of the "elements of programming style" proposed by Kernighan and Plauger (1974) were analyzed.

The first caveat presented is "write clearly - don't be too clever." They illustrate this point by presenting two alternative programs for constructing an identity matrix:

```
      DO 14 I = 1,N
      DO 14 J = 1,N
   14 V(I,J) = (I/J) * (J/I)
```

and

```
      DO 14 I = 1,N
         DO 12 J = 1,N
   12       V(I,J) = 0.0
   14    V(I,I) = 1.0
```

In terms of the propositional descriptions in Figure 1, the first has fewer propositions. Most programmers would agree, however, that the first is more difficult to comprehend. This is due to the number of implicit propositions involved. The first version exploits the properties of integer arithmetic in FORTRAN and requires the reader to recall these properties. The reader must recall, for example, that "I/J" is zero if J is less than I and that "(I/J) * (J/I)" is one if and only if I is equal to J, and zero otherwise. The propositions corresponding to this information are required in order to understand this program. The use of implicit propositions places upon the reader the burden of determining that implicit information is required, retrieving this information, and integrating it with the explicit propositions. Since these propositions must be generated in order to write this program, it seems reasonable to expect that the "clever" program also took longer to write.

We have considered several of the examples presented by Kernighan and Plauger and the programming principles which they advocate. Many of them are interpretable in terms of minimization of the number of propositions and the complexity of the propositional hierarchy. Even more predominant, however, are instances like the above, in which avoidance of implicit propositions appears to be the key. This suggests that one principle of good programming practice would be to eliminate, or at least minimize, the use of implicit propositions. This is consistent with the results of research on text memory.

AN EXPERIMENTAL TEST OF THE PROPOSITIONAL STRUCTURE OF PROGRAMS: AN INVESTIGATION OF DEBUGGING

The present experiment is intended as a preliminary validation of the theoretical framework described in the preceding sections. This experiment was designed as a partial replication of the Gould and Drongowski (1974) experiment, which indicated that debugging performance is related to the type of bug. The present experiment included the location of a bug in the propositional hierarchy as an additional factor. This allows a fairly direct comparison of the hypothesis proposed in this paper with that proposed by Gould and Drongowski. We will also consider the software physics hypothesis (e.g., Gordon and Halstead, 1976) and its usefulness in explaining program comprehension and debugging.

METHOD

Overview

Forty-eight undergraduate students participated in, and successfully completed, the experiment. All participants had at least moderate experience with computer science and FORTRAN. Working individually, each subject attempted to debug two separate FORTRAN programs. Each program contained a single nonsyntactic bug. For each subject, both programs contained the same class of bug (assignment, array, or iteration), but the two bugs differed in their location in the program's underlying propositional hierarchy. The principal dependent measure was the time required to locate each bug. The time required to correct the bug and the times and nature of incorrect indications of a bug's location were also recorded.

Subjects

This experiment used 59 subjects who were recruited through a newspaper advertisement, classroom announcements, and bulletin board notices, and were paid $6.00 for participating in the experiment. Subjects were undergraduates who had completed a minimum of three computer science courses, at least one of which involved FORTRAN, or who had equivalent experience. Subjects were randomly assigned to one of six experimental conditions in order of their arrival for the experiment. The data from eleven participants were excluded from the analysis because they failed to successfully debug either of the experimental programs within a 90-minute time limit. This was an a priori criterion. Whenever a subject was eliminated by this criterion, a new subject was immediately run in the same condition as a replacement. Thus, 48 participants who completed the experiment were evenly divided into the six experimental conditions.

Materials

Two FORTRAN programs were used. These programs, called TALL and CORR, were previously used by Gould and Drongowski (1974) and are modifications of statistical analysis routines contained in the IBM Scientific Subroutine Package.

Each program was modified to include a single bug. Three classes of bug, as defined by Gould and Drongowski, were included: assignment bugs, iteration bugs, and array bugs. Assignment bugs were defined as any error in an assignment statement; array bugs caused an array limit to be exceeded; and iteration bugs caused an incorrect number of iterations through a loop without causing an array limit to be exceeded.

In addition, the location of each bug was varied. These locations were determined on the basis of our presumed propositional representations of these programs. That is, for each class of bug, one bug was placed higher in the propositional hierarchy than the other bug. This resulted in six versions of each of the two programs; two versions for each of the three classes of bug, with the bugs differing in their relative depth in the propositional hierarchy.

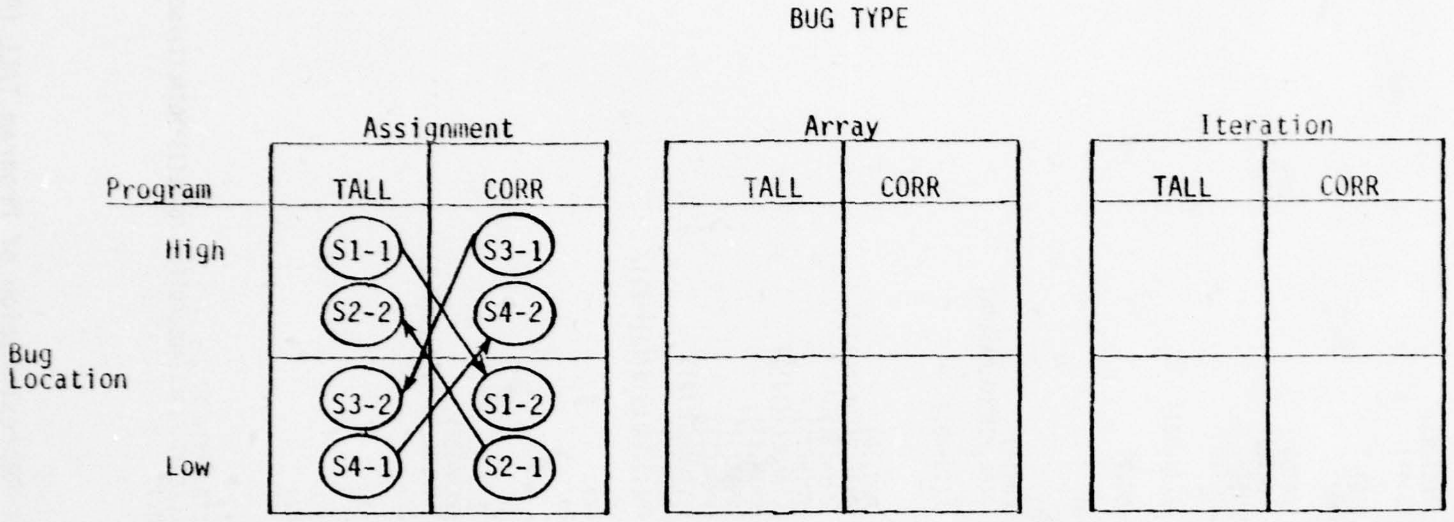
The two programs, TALL and CORR, are presented in Appendix A. The propositions that are considered to underly these programs and the hierarchial structure of these propositions are described in Figures 4 and 5. For simplicity, we have omitted the single top level node that serves to connect the represented sub-hierarchies. We identify with Roman numerals those propositions which correspond to COMMENT statements and which serve primarily to introduce these sub-hierarchies. In our later analyses, we will consider only those propositions, identified with Arabic numerals, which represent the actual code and which the programmar must comprehend and integrate. The bugs that were employed are described in Table 2. Each program was compiled with a Prime 400 FORTRAN compiler to eliminate any syntactic errors due to modification.

Design

Three variables were involved in this study -- program, bug type, and bug location. Because we were interested primarily in observing initial comprehension effects, it was necessary to present each program only once to a given participant. Therefore, bug type was varied between groups of participants, so that there was an assignment group, an array group, and an iteration group. Within each of these groups, program and bug location were varied in a Latin square design. Thus, a participant in the assignment group might be given program TALL with a bug low in the propositional hierarchy and program CORR with a high bug. Another participant in the assignment group might be given the combination TALL-high and CORR-low.

The experimental design is described in Table 1. We have illustrated the order of program presentation for four subjects in the assignment group. Subject 1 (S1) debugged TALL-high followed by CORR-low, while Subject 2 received these programs in the reverse order. Subject 3 debugged CORR-high followed by TALL-low and this order was reversed for Subject 4. That is, within each bug-type group (assignment, array, iteration), program presentation order (TALL, CORR) and bug-location (high, low) were counterbalanced. Although not shown in Table 1, similar orders were presented in the array and iteration groups.

Table 1. Experimental Design



		BUG TYPE					
		Assignment		Array		Iteration	
	Program	TALL	CORR	TALL	CORR	TALL	CORR
Bug Location	High	S1-1 S2-2	S3-1 S4-2				
	Low	S3-2 S4-1	S1-2 S2-1				

```
1.  SET(NV,20)
2.  SET(NO,30)
3.  SET(IER,0)
4.  INITIALIZE(VECTORS)
5.  SIZE(VECTORS,NV)
6.  SAME(NV,1)
7.  SET(TOTAL,0.0)
8.  SET(AVER,0.0)
9.  SET(SD,0.0)
10. SET(VMIN,1.0E10)
11. SET(VMAX,-1.0E10)
12. SET(SCNT,0.0)
13. DO(J,1,NO,α)
14. SAME(NO,2)
15. SET(IJ,VALUE(IJ-NO))
16. SAME(NO,2)
17. RANGE(IJ,1-NO,0)
18. TEST(S(J))
19. COND(S(J)=0)
20. CONTINUE
21. COND(S(J)≠0)
22. INCREMENT(SCNT,1)
23. CALCULATE(TOTAL,MINIMA,MAXIMA)
24. DO(I,1,NV,β)
25. SET(IJ,VALUE(IJ+NO))
26. SAME(NO,2)
27. SAME(IJ,15)
28. RANGE(IJ,1,600,30)
29. INCREMENT(TOTAL,VALUE(A(IJ))
30. TEST(A(IJ),VMIN(I))
31. COND(A(IJ)<VMIN(I))
32. SET(VMIN(I),VALUE(A(IJ)))
33. COND(A(IJ)>VMIN(I))
34. TEST(A(IJ),VMAX(I))
35. COND(A(IJ)>VMAX(I))
36. SET(VMAX(I),VALUE(A(IJ))
37. COND(A(IJ)<VMAX(I))
38. INCREMENT(SD(I),VALUE(A(IJ)**2))
39. TEST(SCNT)
40. SAME(SCNT,22)
41. COND(SCNT≤0)
42. SET(IER,1)
43. END
44. COND(SCNT>0)
45. DO(I,1,NV,δ)
46. CALCULATE(AVERAGES)
47. SET(AVER(I),VALUE(TOTAL(I)/SCNT))
48. SAME(SCNT,22)
49. TEST(SCNT)
50. COND(SCNT=1)
51. SET(IER,2)
52. SET(SD,0.0)
53. NUMBER(SD,NV)
54. SAME(NV,1)
55. END
56. COND(SCNT≠1)
57. CALCULATE(STD.DEV's.)
58. SET(SD,FCALL(SQRT,FCALL(ABS,SD(I))-TOTAL(I)*TOTAL(I)/SCNT)/(SCNT-1.0))))
59. NUMBER(SD,NV)
60. SAME(NV,1)
61. END
```

Figure 4. Propositional Representation of Program TALL (Page 1 of 2)

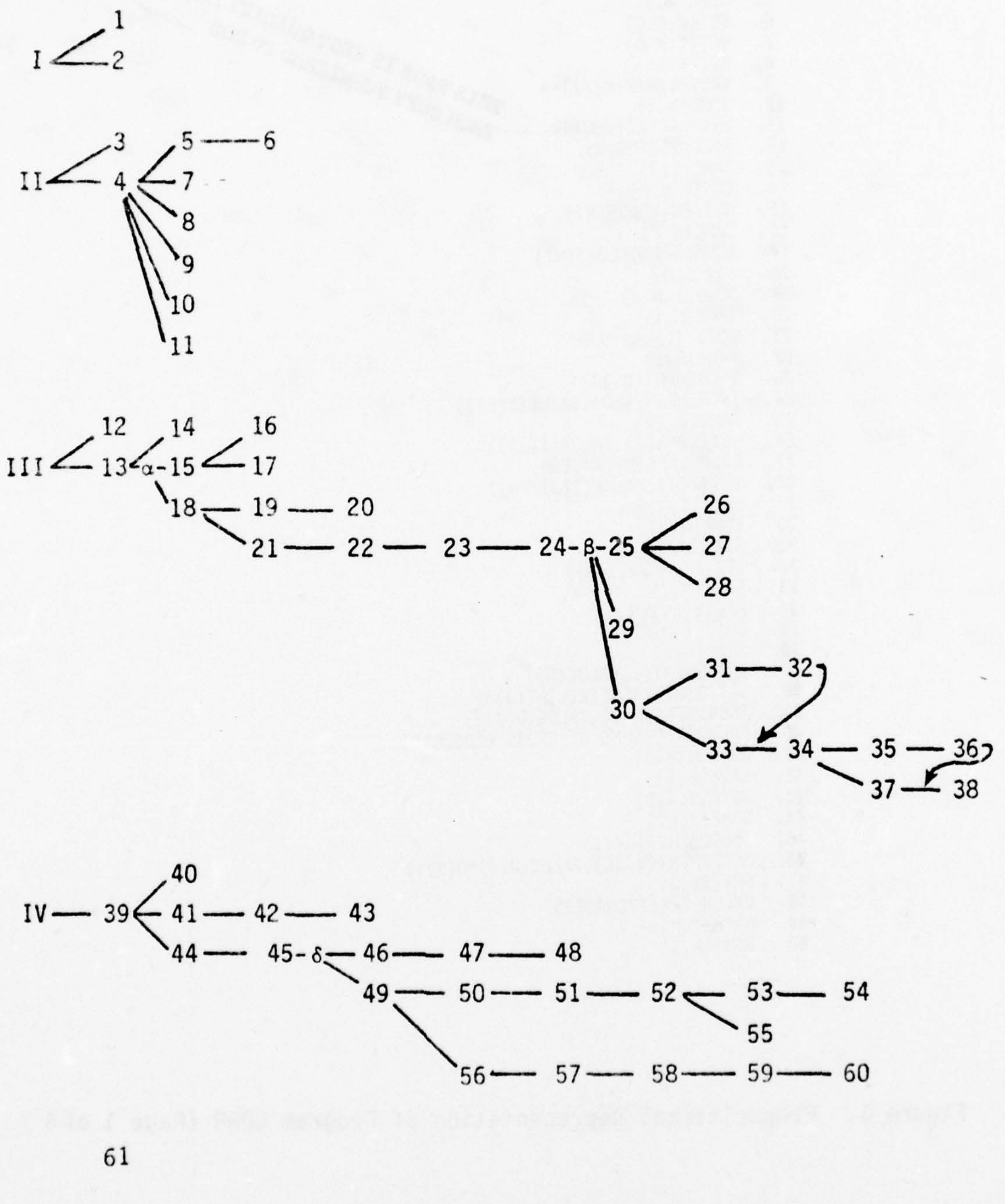


Figure 4. Propositional Representation of Program TALL (Page 2 of 2)

```
1.  SET(M,20)
2.  SET(N,30)
3.  INITIALIZE(VECTORS)
4.  SIZE(VECTORS,M)
5.  SAME(M,1)
6.  SET(B,0.0)
7.  SET(T,0.0)
8.  SET(K,α)
9.  VALUE((M*M+M)/2)=α
10. SAME(M,1)
11. INITIALIZE(VECTOR)
12. SIZE(VECTOR,K)
13. SAME(K,8)
14. SET(R,0.0)
15. SET(FN,VALUE(N))
16. SAME(N,2)
17. EQUALS(FN,FLOAT(N))
18. SET(L,0)
19. DO(J,1,M,α)
20. SAME(M,1)
21. DO(I,1,N,β)
22. SAME(N,2)
23. INCREMENT(L,1)
24. INCREMENT(T(J),VALUE(X(L)))
25. SAME(J,19)
26. SET(XBAR(J),VALUE(T(J)))
27. CALCULATE(AVERAGES)
28. SET(T(J),VALUE(T(J)/FN))
29. DO(I,1,N,γ)
30. SAME(N,2)
31. SET(JK,0)
32. SET(L,VALUE(I-N))
33. SAME(N,2)
34. RANGE(L,1-N,0)
35. DO(J,1,M,δ)
36. SAME(M,1)
37. INCREMENT(L,VALUE(N))
38. SET(D(J),VALUE(X(L)-T(J)))
39. INCREMENT(B(J),VALUE(D(J)))
40. CALCULATE(SUMS OF CROSS PRODUCTS)
41. DO(J,1,M,α)
42. SAME(M,1)
43. DO(K,1,J,B)
44. SAME(J,41)
45. INCREMENTS(JK,1)
46. INCREMENT(R(JK),VALUE(D(J)*D(K)))
47. SET(JK,0)
48. CALCULATE(AVERAGES)
49. DO(J,1,M,α)
50. SAME(M,1)
```

Figure 5. Propositional Representation of Program CORR (Page 1 of 4)

```
 51.  SET(XBAR(J),VALUE(XBAR(J)/(FN))
 52.  CALCULATE(ADJUSTED SUM OF CROSS PRODUCTS)
 53.  DO(K,1,J,β)
 54.  SAME(J,49)
 55.  INCREMENT(JK,1)
 56.  SET(R(JK),VALUE(R(JK)-B(J)*B(K)(FN))
 57.  SET(JK,0)
 58.  CALCULATE(STD.DEV'S.)
 59.  DO(J,1,M,α)
 60.  SAME(M,1)
 61.  INCREMENT(JK,VALUE(J))
 62.  SET(STD(J),FCALL(SQRT,FCALL(ABS,R(JK))))
 63.  CALCULATE(R VALUES)
 64.  DO(J,1,M,α)
 65.  SAME(M,1)
 66.  DO(K,J,M,β)
 67.  SAME(J,64)
 68.  SET(JK,VALUE(J+(K*K-K)/2))
 69.  SET(L,VALUE(M*(J-1)+K))
 70.  SET(RX(L),VALUE(R(JK))
 71.  SAME(JK,68)
 72.  SAME(L,69)
 73.  SET(L,VALUE(M*(K-1)+J))
 74.  SAME(M,1)
 75.  SET(RX(L),VALUE(R(JK)))
 76.  SAME(JK,68)
 77.  SAME(L,73)
 78.  TEST(STD(J)*STD(K))
 79.  COND(STD(J)*STD(K)=0)
 80.  EITHER(STD(J)=0,STD(K)=0)
 81.  SET(R(JK),0.0)
 82.  SAME(JK,68)
 83.  CONTINUE
 84.  COND(STD(J)*STD(K)≠0)
 85.  NEITHER(STD(J)=0,STD(K)=0)
 86.  SET(R(JK),VALUE(R(JK)/(STD(J)*STD(K))))
 87.  SAME(JK,68)
 88.  CONTINUE
 89.  SET(FN,FCALL(SQRT,VALUE(FN-1.0)))
 90.  CALCULATE(STD)
 91.  DO(J,1,M,α)
 92.  SAME(M,1)
 93.  SET(STD(J),VALUE(STD(J)/FN))
 94.  SET(L,VALUE(-M))
 95.  NEGATIVE(L,M)
 96.  DO(I,1,M,α)
 97.  SAME(M,1)
 98.  SET(L,VALUE(L+M+1))
 99.  RANGE(L,1,(M-1)*(M+1),M+1)
100.  SET(B(I),VALUE(RX(L))
101.  END
```

Figure 5. Propositional Representation of Program CORR (Page 2 of 4)

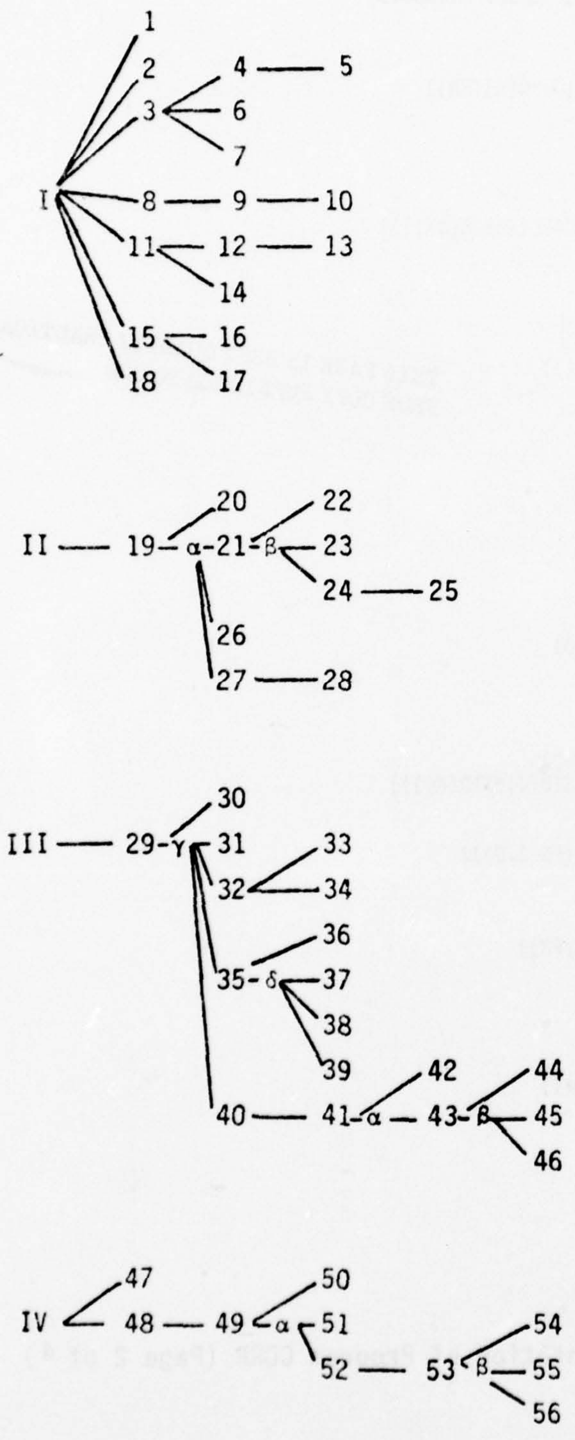


Figure 5. Propositional Representation of Program CORR (Page 3 of 4)

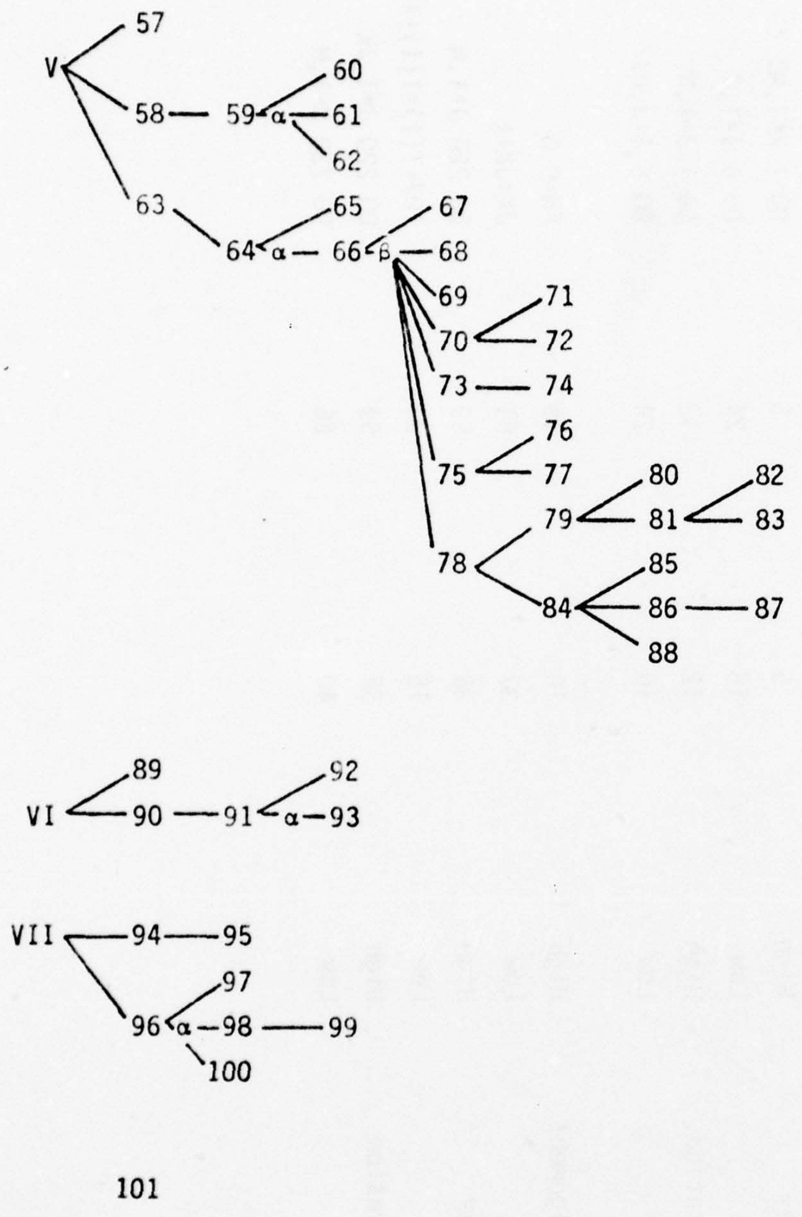


Figure 5. Propositional Representation of Program CORR (Page 4 of 4)

Table 2. A Description of the Bugs Used in the Experiment

Program	Bug Type	Bug Location	Line Number	Proposition Number	Modification
TALL	Assignment	High	11	12	SCNT=NO
		Low	17	25	IJ=IJ+I
	Array	High	5	5	DO 1 K=1,NO
		Low	16	24	DO 6 I=1,J
	Iteration	High	12	13	DO 7 J=1,NV
		Low	16	24	DO 6 I=J,NV
CORR	Assignment	High	10	15	FN=0.0
		Low	37	61	JK=JK+1
	Array	High	36	59	DO 220 J=1,N
		Low	15	24	107 T(I)=T(I)+X(L)
	Iteration	High	36	59	DO 220 J=1,JK
		Low	40	66	DO 230 K=1,M

Procedure

In all respects, the procedure employed was intended to replicate that used by Gould and Drongowski (1974). Each participant was given two program listings, one at a time. Participants were told that: (1) the program is syntactically correct, (2) the program contains a single non-syntactic error in one line, (3) the input/output statements were removed from the programs, but the data can be assumed to be stored in the computer correctly, and (4) they should try to find the bug as quickly and accurately as possible and to indicate to the experimenter the line that contains the error. Participants were allowed a rest break between programs, but were instructed to continue to work on a program, once they had started, until they found the bug.

Participants were timed with a stopwatch and times were recorded to the nearest hundredth of a minute. When a participant indicated that he or she had found the bug, timing was stopped. If the participant was incorrect, the time and indicated location were recorded, the participant was informed that he or she was incorrect and asked to continue, and timing was restarted; if the participant was correct, the time was recorded. Timing was also stopped whenever a participant had a question about the experimental procedure. After a participant correctly indicated the location of the bug, he or she was asked to rewrite the erroneous statement in order to eliminate the error; times and incorrect responses were recorded.

Participants were allowed 45 minutes in which to find the bug. If the bug was not found within this time, the location and nature of the bug were explained to the participant. Participants were permitted to quit debugging a program if they had not found the error within 15 minutes, although they were not explicitly informed of this possibility beforehand. The times for such participants were scored as 45 minutes.

This variable time criterion was originally used in the experiment reported by Gould and Drongowski (1974). A debugging task can be very demanding and participants who expressed extreme frustration were allowed to quit before the 45-minute time limit was reached. In the present experiment, three participants were allowed to quit debugging at their request, before the 45-minute time limit; the actual times were 36.6, 41.5, and 43.4 minutes.

RESULTS

The design of this experiment permits several types of analysis. Recall that in the Gould (1973) and Gould and Drongowski (1974) studies each subject attempted to debug a given program (with a different bug each time) three separate times. Although program comprehension is an important component of measured debugging time on the first presentation of a program, this effect is reduced on subsequent presentations. Since program comprehension is of principal interest in the present study, a design was adopted to minimize the effects of learning or debugging skills and processes and to highlight the program comprehension process.

In the present design, each participant debugged two separate programs. Although this may lessen the effect of learning, it may not entirely eliminate it. Since both programs were statistical in nature, it may be the case that comprehension of the second program is facilitated by similarities between the two programs. In addition, although program and bug location were treated as within-subjects variables, bug type was treated as a between-subject variable; that is, both programs that a participant attempted to debug contained the same type of bug. If subjects rapidly learn to adopt a debugging strategy that is oriented toward detecting a given type of bug, this may also facilitate performance on the second program and reduce the effects of program comprehension.

Such a repeated-measures design offers several advantages. Such designs increase statistical power without requiring an increase in sample size and they allow for better control of individual differences than do factorial designs. However, in those cases in which there may be differential effects of the factors in terms of practice, transfer of training, etc., a factorial design is preferred.

An experimental design was selected, therefore, that could be treated as a repeated measures design, but if this analysis was determined to be inappropriate, could also be treated as a factorial design. The results of this experiment are analyzed both as a repeated measures design including both programs debugged by a subject and as a factorial design in which only the first program is considered.

Overall Debug Times

We will consider first the overall design, which includes both programs presented to participants. The means and standard deviations of the times to locate bugs are shown in Table 3. Although debug times were measured to the nearest hundredth of a minute, times were transformed to the nearest twentieth of a minute for analysis in order to minimize any experimenter-induced variability in time measurement, but still maintain sufficient detail for a meaningful analysis. Since not all participants were able to locate both bugs, debug times are positively skewed. Subsequent analyses, therefore, are based on logarithmic transforms. The results of the analysis of variance are shown in Table 4.

All three main effects were highly significant. For program, CORR was more difficult to debug than TALL (F(1, 42)=11.44, p<.005). The bug location factor was significant, with bugs low in the propositional hierarchy being more difficult to find than bugs placed relatively high (F(1,42)=11.45, p<.005). A significant effect was also observed for bug type (F,(2,42)=10.91, p<.001). In order to determine the relative difficulty

Table 3. Means and (Standard Deviations) of Debug Times

		BUG TYPE						
		Assignment		Array		Iteration		
Program		TALL	CORR	TALL	CORR	TALL	CORR	
Bug Location	High	13.21	22.59	11.85	31.32	21.64	23.59	20.70
		(9.85)	(14.90)	(15.06)	(13.55)	(8.60)	(10.99)	(13.49)
	Low	26.06	45.00	24.62	10.50	28.44	43.74	29.73
		(16.70)	(0.)	(13.63)	(8.37)	(9.14)	(3.57)	(15.40)
		26.71		19.58		29.35		
		(16.52)		(15.14)		(11.97)		

Program	TALL	CORR
	20.97	29.46
	(13.47)	(15.56)

Table 4. Summary Table for Debug Times

Source	df	Mean Square	F	
Between Subjects				
Bug Type (T)	2	.9696	10.91	p<.001
P X L	1	.4683	5.27	p<. 05
T X P X L	2	.9464	10.65	p<.001
Subjects within groups	42	.0889		
Within				
Program (P)	1	.9392	11.44	p<.005
Bug Location (L)	1	.9410	11.46	p<.005
P X T	2	.0829	1.01	
L X T	2	.2113	2.57	p<. 10
Residual	42	.0821		

of each of the three bug types, a Tukey (a) procedure was used to test differences between treatment means. No differences were observed between assignment and iteration bugs, but both were significantly more difficult to detect ($p<.01$) than array bugs.

A significant interaction was observed between program and bug location ($F(1,42)=5.27$, $p<.05$). The nature of this interaction is most clearly illustrated by considering the order in which the programs were presented to participants. Recall that half of the participants in each condition first attempted to debug TALL and half were first given CORR. In Figure 6, we present debug time as a function of program, order of program presentation, and bug location. It can be seen that bug location has significant effects on program TALL when this program is presented first, with bugs placed relatively low being more difficult to detect than bugs placed relatively high, and that CORR, when presented first, is relatively unaffected by bug location. When we consider only the second program that was presented to participants, however, these effects are reversed.

The three way interaction between program, bug type, and bug location was significant ($F,(2,42)=10.65$, $p<.001$). Notice that the low array bug in CORR was detected more quickly than the bug placed higher in the hierarchy and was also detected more quickly than the corresponding bug in TALL. This condition is, in fact, the only instance in which a low bug was detected more quickly, on the average, than the corresponding high bug, and it is the only instance in which a bug in CORR was detected more quickly than the corresponding bug in TALL. This interaction is primarily due to the fact that the time to locate array bugs did not vary consistently as a function of either program or bug location.

In interpreting these results, it should be noted that no participant found the low assignment bug in CORR. A reconsideration of the bug induced in this condition indicates that it is, indeed, a nonsyntactic error in one line of the program which prevents successful computation of the required result. It is possible that this bug did not prevent

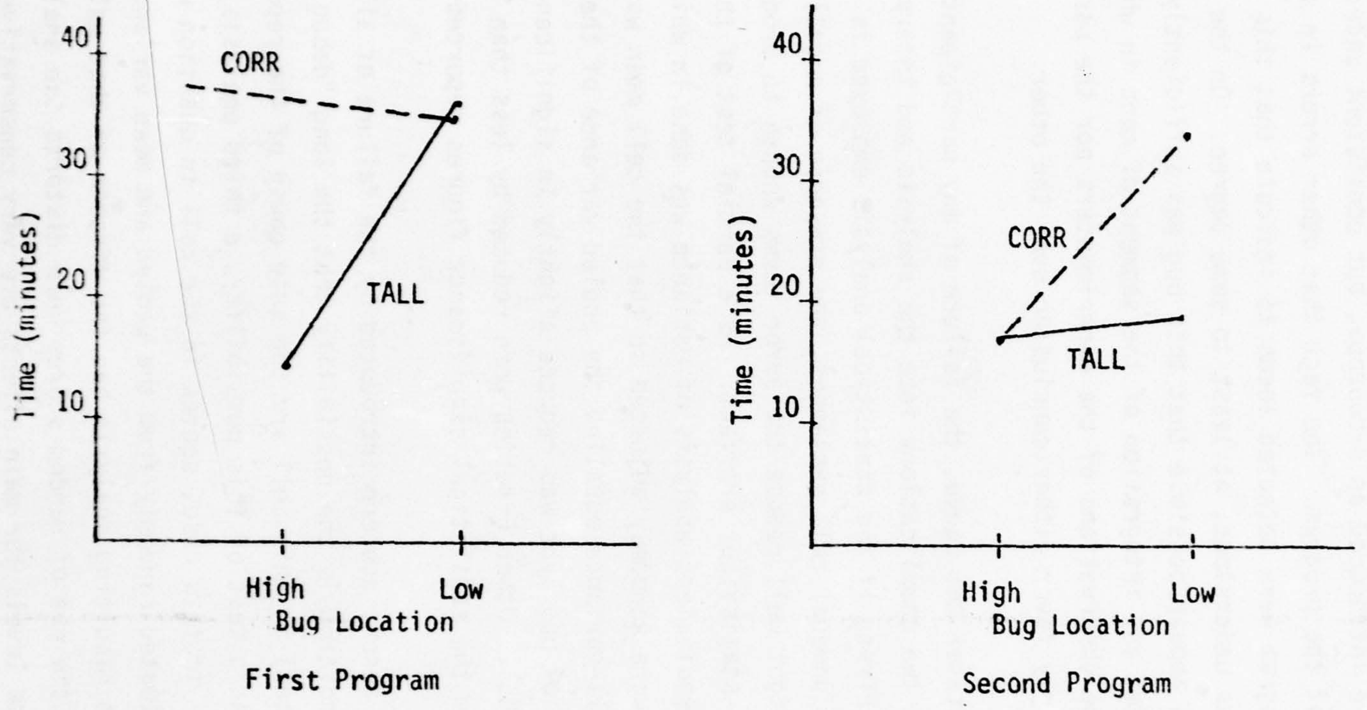


Figure 6. Mean Debug Time as a Function of Program, Program Presentation Order, and Bug Location

comprehension of the segment of the program in which it occurred, although it may have introduced an erroneous, but consistent understanding of the function of the program. The fact that other errors in surrounding lines of the program were detected seems to indicate that this segment of the program was understood, at least to some degree. On the other hand, it would also appear possible that this bug was sufficiently disruptive that it prevented any integration of the segment of code in which it appeared. Neither the observations of the experimenters nor the participants' error data strongly favor either conclusion over the other.

Whatever the cause, the failure of any participant to find this bug introduced two complications into the analysis and interpretation of the results. First, if the statistical analysis employed is not robust with respect to unequal cell variances, the presence of a cell with zero variance might well reduce the error terms enough to produce significant results by statistical artifact. As a partial test of this possibility, a second, equivalent analysis of variance was done in which the scores in this cell were randomly adjusted so that the cell mean was not changed, but the cell variance equalled the pooled variance of the other cells. The effect of bug type was reduced slightly in significance ($F(2,42)= 7.92$, $p<.005$). Other F-ratios were reduced by less than 15%, with no change from the statistical significance figures reported in Table 4.

The second concern introduced by the failure of all participants in this condition is the possibility that the long "debug times" (estimated as 45 minutes) in this cell are the sole cause of the reported significant effects. As a test of this possibility, a third analysis of variance was performed. In this case, scores in the cell in question were replaced by scores estimated randomly from the pooled and mean variance of the other cells. The resulting analysis has the effects of that cell entirely removed. Obviously, the use of random scores here distorts the analysis, and the significance levels for main effects are very conservative. If the main effects remain significant in this analysis, it is reasonable to conclude that the cell in question is not the sole cause of the significant results. In fact, all main effects are reduced in magnitude, but remain significant

Table 5. Means and (Standard Deviations) of Debug Times for First Program

		BUG TYPE						
		Assignment		Array		Iteration		
Program		TALL	CORR	TALL	CORR	TALL	CORR	
Bug Location	High	12.91 (9.26)	32.90 (15.12)	9.75 (9.11)	42.08 (5.85)	19.58 (9.54)	30.14 (12.04)	24.56 (14.97)
	Low	34.80 (20.40)	45.00 (0.)	34.64 (10.49)	12.95 (11.64)	33.58 (5.98)	42.48 (5.05)	33.91 (14.32)
		31.40 (17.04)		24.85 (16.61)		31.44 (11.45)		

Program	TALL	CORR
	24.21	34.26
	(14.94)	(14.10)

Table 6. Summary Table for Debug Times for First Program

Source	df	Mean Square	F	
Program (P)	1	.6541	6.89	p<.025
Bug Location (L)	1	.4902	5.17	p<. 05
Bug Type (T)	2	.2240	2.36	
P X L	1	.9729	10.25	p<.005
P X T	2	.0601	< 1	
L X T	2	.0567	< 1	
P X L X T	2	.4952	5.22	p<. 01
Residual	36	.0949		

(bug type, $F(2,42)=8.75$, $p<.01$; program, $F(1,42)=4.66$, $p<.05$; bug location $F(1,42)=4.68$, $p<.05$).

While the "ceiling effect" observed in this experimental condition did introduce statistical difficulties., it is worth noting that it also reduced the magnitude of the mean differences due to bug location, etc. Had participants been allowed to continue, the observed debug times in this condition would have been greater (probably much greater) than those reported here.

The pattern of results reported in Table 4 appears to be a substantially correct representation of the effects of the experimental variables under the conditions of the study.

First Program Debug Times

The data were also analyzed with respect to only the first program that each participant attempted to debug. The resulting factorial design has the advantage that it eliminates any spurious differences that are due to practice, transfer of training, etc., and allows for a more direct comparison of the effects of bug type and of bug location. The means and standard deviations for each condition are presented in Table 5. The results of this analysis are summarized in Table 6.

Significant effects were observed due to program ($F(1,36)=6.89$, $p<.005$) and due to bug location ($F(1.36)=5.17$, $p<.05$). As in the previous analysis, CORR was more difficult to debug than TALL and bugs that were relatively low in the hierarchy were more difficult to detect than those that were higher. The effect due to bug type, however, was not significant.

A significant interaction was observed between program and bug location ($F(1,36)=10.25$, $p<.005$). A possible explanation of this interaction, which will be discussed in detail below, is that these factors are not totally independent since the structure of the program determines the bug locations that can be selected. The three-way interaction between bug type, bug location, and program was also significant (F(2,36)=5.22, p<.01).

Although bug type has a significant effect overall, it has no apparent effect on the first program that participants attempted to debug. This suggests that experience with the first program led subjects to adopt a more efficient strategy for searching for a given class of bug in the second program and that any strategies developed were not equally effective for each of the three types of bugs used. Since both programs presented to a participant contained the same class of bug, the use of such strategies could produce large performance differences in the overall results.

Second Program Debug Times

Recall that one possible explanation of the Gould (1973) and Gould and Drongowski (1974) finding, that array and iteration bugs are easier to detect than assignment bugs, is that array and iteration bugs may be detectable on the basis of a less detailed understanding of the program than is required to detect assignment bugs. Specifically, a subject who adopts a strategy of comparing loop indices with dimensioned array sizes might be able to detect array and iteration bugs without thoroughly comprehending the program.

The analysis presented above implies that this is not the case for the first program presented to subjects, since bug type did not produce significant performance differences. If we consider only the second program the participants attempted to debug, however, bug type produces a highly significant effect (F(1,36)=8.69, p<.001). (See Tables 7 and 8.) In Table 7, we also present the percentage decline in debug times over repeated trials for each of the three factors studied. For example, the mean debug time for program TALL was 24.21 minutes when it was presented first and 17.73 minutes when it was presented second; this is a decline of 27%. Overall, the mean decline in debug time attributable to practice effects is 31%. In general, the improvement on each level of the three factors is fairly close to this mean value. The single exception is with array bugs. In this case, the decline in debug time due to practice is 42%.

It appears, therefore, that subjects who located an array bug in the first program adopted a strategy of initially searching for an array bug in the second program, and due to the experimental design, this strategy often proved successful. The overall effect of bug type, therefore, may be due to an interaction between debugging strategies and the experimental design employed rather than any direct effect of bug type.

ADDITIONAL ANALYSES

The large number of significant effects observed (see Table 9), as well as the apparently unequal transfer effects, suggest that additional analyses would be informative. It is highly probable that the three factors involved in this study -- bug type, bug location, and program -- are not completely independent. This problem is also apparent in the Gould (1973) and Gould and Drongowski (1974) studies of debugging and is probably a common problem in all multi-factor studies of programming activities. This problem, however, appears to be due to the nature of the tasks studied rather than to a lack of experimental rigor.

Consider, for example, assignment bugs which are loosely defined as "errors in assignment statements." Within a given program, we could place assignment bugs in several distinct statements. These statements would differ with respect to several potentially relevant factors such as statement length, serial position of the statement, whether the error is on the right or left side of the assignment operator, etc. The nature of a program may prevent accurately controlling for the effects of these factors if assignment statements are not uniformly distributed as a function in serial position, depth in the program structure, etc. When additional experimental factors are introduced, this problem becomes increasingly more complicated.

Each of the three factors involved in this experiment may significantly affect the other two factors. For example, the selection of a given program dictates, in part, the hierarchial level at which bugs can be placed and the exact form of the bug. In order to reduce such effects, we will consider each experimental factor separately and determine the extent to which each factor is responsible for the observed results.

Table 7. Means and (Standard Deviations) of Debug Times for Second Program

		BUG TYPE						
		Assignment		Array		Iteration		
Program		TALL	CORR	TALL	CORR	TALL	CORR	
Bug Location	High	13.5 (11.84)	12.29 (2.50)	13.95 (20.85)	20.58 (9.27)	23.71 (8.38)	17.04 (4.72)	16.84 (31%) (10.80)
	Low	17.31 (5.54)	45.00 (0.)	14.61 (7.48)	8.05 (3.48)	23.30 (9.43)	45.00 (0.)	25.54 (25%) (15.58)
		22.02 (30%) (15.06)		14.30 (42%) (11.78)		22.14 (30%) (12.56)		

Program	TALL	CORR
	17.73 (27%) (11.22)	24.66 (28%) (15.75)

Table 8. Summary Table for Debug Times for Second Program

Source	df	Mean Square	F	
Program (P)	1	.3156	4.36	$p<.05$
Bug Location (L)	1	.4513	6.23	$p<.025$
Bug Type (T)	2	.6294	8.69	$p<.001$
P X L	1	.0004	< 1	
P X T	2	.0277	< 1	
L X T	2	.1701	2.35	$p>.10$
P X L X T	2	.4622	6.38	$p<.005$
Residual	36	.0724		

Table 9. Summary of Results

	Significance Level		
Source	Overall Debug Time	First Program Debug Time	Second Program Debug Time
Bug Type (T)	$p < .001$		$p < .001$
Program (P)	$p < .005$	$p < .025$	$p < .05$
Bug Location (L)	$p < .005$	$p < .05$	$p < .025$
P x L	$p < .05$	$p < .005$	
P x T			
L x T	$p < .10$		
P x L x T	$p < .001$	$p < .01$	$p < .005$

Programs

Overall, the program factor was highly significant ($F(1,42)=11.44$, $p<.005$), with CORR being more difficult than TALL. In addition, in the six comparisons that can be made between these two programs, at equivalent levels of the other two factors, CORR is more difficult in five cases. (See Table 3.)

An examination of the listings of these two programs reveals several obvious differences. One such difference is the relative emphasis on the statistical concepts involved in these programs. It seems probable that these students would generally be more familiar with the concepts of "minima" and "maxima" involved in TALL than with the "correlation coefficients" and "sums of cross-products" involved in CORR.

Another difference, about which participants frequently commented, concerns the use of meaningful variable names. While CORR uses a large number of one- and two-character variable names (e.g., R, RX, and B), the variable names used in TALL (e.g., TOTAL, VMAX, and AVER) were probably more meaningful to the participants. Perhaps the most obvious difference, though, is the fact that CORR contains more statements (and propositions) than TALL. If propositions are processed at a constant rate, CORR would take longer to comprehend.

Although these, and similar, variables may account for, or at least contribute to, the observed differences between programs, the present experiment does not allow an objective analysis of such variables. There is, however, a theory that attempts to predict program complexity and debugging time solely as a function of the program factor and we will consider it briefly. The "software physics hypothesis" (e.g., Gordon and Halstead, 1976) involves the use of measurable program properties to calculate the number of "elementary mental discriminations" involved in writing or comprehending a program.

Applying the rules described by Bulut, Halstead, and Bayer (1974), we calculate that TALL involves approximately 29,000 elementary mental discriminations and CORR involves approximately 105,000. Although the software physics hypothesis appears able to explain the observed order of programs with respect to overall debug time, it cannot explain the magnitude of the difference between observed debug times. The software physics hypothesis suggests that debug time is a linear function of the number of elementary discriminations. The ratio of the number of elementary mental discriminations in CORR to that in TALL is 3.62. The ratio of the number of propositions in each program is 1.66, which is much close to the observed ratio of 1.44 for observed debug times. It is also not clear how this hypothesis could account for differences due to bug type and bug location. For example, Love and Bowman (1976) applied the software physics hypothesis to a reanalysis of a debugging study reported by Gould (1975). Although they were able to account for certain aspects of program differences, they did not consider the fact that different bugs within a given program are not equally difficult to detect. If the software physics hypothesis is to be considered a valid explanation of program complexity, it must be expanded to account for such results.

Bug Type

That the class of bug (array, assignment, or iteration) affects debugging time was suggested by Gould (1973) and Gould and Drongowski (1974). In both studies, it was concluded that assignment bugs were significantly more difficult to detect than either array or iteration bugs. The present study, however, contradicts this conclusion.

Considering only the first program presentation, there were no significant differences due to bug type. Overall, array bugs were significantly easier to detect than either assignment or iteration bugs. This is probably due to unequal transfer effects among types of bugs. The difference between our results and those reported by Gould and by Gould and Drongowski, with respect to the relative difficulty of types of bug, will be considered in more detail later and described in terms of a bug's location on the propositional hierarchy.

Although the class of bug may or may not affect debugging performance, bug type cannot account for either program differences or differences due to bug location. As noted above, bugs in a given class are not necessarily equal. For example, an assignment bug has a different form in each program and assignment bugs within a given program can be placed at different levels in the underlying propositional hierarchy. While neither the bug type nor program factors can account for the effects attributed to the other two factors, the analysis of the bug location factor appears more promising.

Bug Location

The propositional representations that we assume to underly the programs TALL and CORR are described in Figures 4 and 5, respectively. Within these representations, there are two basic methods for specifying bug location. We can consider either the level in the hierarchy at which the bug appears or we can consider the number of propositions that precede the bug. In the remainder of this section, we will refer to these measures as "depth" and "serial position," respectively.

Our original expectation was that level in the hierarchy (depth) was the more interesting metric. Although we also expected some serial position effect, we assumed that this effect would be fairly small, and we selected bug location primarily on the basis of depth. A subsequent analysis of the propositional hierarchies, however, indicates that our placement of bugs caused depth and serial position to be highly correlated. That is, bugs that are at lower levels are, in this study, preceded by more propositions than are bugs at higher levels. The single exception occurs in the array bugs placed in program CORR; the low bug is located in proposition number 24 and the high bug in number 59.

In Table 10, we present the mean times required to locate the bug in each experimental condition, the level of the bug, the number of the proposition that contains the bug (i.e., the number that precede the bug plus the proposition including the bug), and the number of cases in which

Table 10. Propositional Descriptions of Bugs and Mean Debug Times

Program	Bug Type	Bug Location	Level	Proposition Number	Overall Mean Debug Time	First Program Mean Debug Time	Number of Bugs Not Found
TALL	Assignment	High	1	12	13.21	12.91	0
		Low	7	25	26.06	34.8	3
	Array	High	2	5	11.85	9.75	1
		Low	6	24	24.63	34.64	1
	Iteration	High	1	13	21.64	19.58	0
		Low	6	24	28.44	35.58	0
CORR	Assignment	High	1	15	22.59	32.90	1
		Low	3	61	45.0	45.0	8
	Array	High	2	59	31.33	42.08	3
		Low	3	24	10.50	12.95	1
	Iteration	High	2	59	23.59	30.14	1
		Low	3	66	43.74	42.48	7

the bug was not found. Also presented are both the overall mean debug times and the mean debug times for only the first program presented to subjects.

Before considering these results, we will briefly consider two factors that affect the interpretation of these results. The first con- concerns differences in the propositional hierarchies that underlie the two programs. As can be seen in Figures 4 and 5, program CORR involves a relatively large number of propositions, but comparatively few levels, while program TALL involves a larger number of levels but fewer propositions.

This factor affected our placement of bugs within these programs. While in TALL we could place "high" bugs at levels 1 and 2 and "low" bugs at levels 6 and 7, "high" bugs in CORR are at levels 1 and 2 and "low" bugs only at levels 2 and 3 (see Table 10). Similarly, the serial positions at which bugs were placed varies more widely in CORR than in TALL.

The second factor is concerned with how we should interpret any effects due to either depth or serial position. Theoretically, depth and serial position should be relatively independent and they should affect different aspects of behavior. Actually, however, depth and serial position of bugs may be highly correlated in individual programs.

Our earlier review of text memory research indicated that the number of propositions that must be processed, which is considered here to be serial position, is a direct determinant of reading time. That is, the number of propositions that must be processed in order to locate a bug should be strongly related to the time a programmer spends on comprehending the program.

It is also apparent in text memory research that the probability of recalling a given proposition is a function of the level in the hierarchy at which it appears. This implies that a bug low in the hierarchy will require more processing by a programmer to successfully integrate the corresponding proposition into the programmer's memory structure than a bug that is at a higher level. Serial position, therefore, would be expected to primarily effect comprehension time while depth would primarily effect processing time and the probability of successful integration.

The number of propositions that precede the bug appears to account for observed debug times better than the depth measure. This measure is independent of the class of bug and can account for much more of the observed variance than can the bug type factor. This propositional measure is not, however, independent of the program factor. As can be seen in Figures 4 and 5, these programs involve widely different numbers of propositions (61 in TALL and 101 in CORR). This suggests that the observed differences in programs may be due, in part, to this fact.

Although we initially expected some effect due to serial position, we did not anticipate that this effect would be so large. It is important to note, however, that this is not a simple reading-time phenomenon due to different numbers of propositions and different serial positions of bugs. While conducting this experiment, we informally observed that subjects made several passes over a program over a fairly long period of time. This is consistent with the observations of Gould (1973) and Gould and Drongowski (1974). We initially expected to observe multiple passes and assumed that this would tend to reduce serial position effects. The magnitude of the serial position effect, in spite of multiple passes, led to the consideration of additional hypotheses. From research on text memory, we know that more processing effort is required to integrate propositions at lower levels in the hierarchy than those at higher levels.

In the programs used in this study, the level at which a bug occurs is correlated with the number of propositions that precede the bug. These correlations are .46 for CORR and .89 for TALL. As can be seen in Figures 4 and 5, the propositional hierarchies that were constructed to represent these two programs are roughly triangular in shape. That is, the first propositions in each program are at relatively high levels and the later propositions tend to be at lower levels. Such a correlation may be fairly typical of programs in general.

This triangular structure, however, is probably most pronounced in relatively short programs, such as those used here, and may tend to disappear in longer programs. The maximum number of levels that can be included in a program is, most likely, determined by a programmer's memory and comprehension abilities and also, possibly, by various aspects of the programming language used. In longer programs, therefore, depth might be expected to stabilize after some point rather than continuing to increase.

Returning to our analysis of the effects of bug location, it is reasonable to expect that both depth and serial position would correlate with observed performance, since program debugging involves both comprehension and recall, the latter in the sense of being able to retrieve relevant information from the existing memory structure. As can be seen in Table 10, there is very little correlation between overall mean debug time and depth ($r=.22$). Considering the individual programs, however, this correlation is .33 for CORR and .80 ($p<.05$) for TALL. When we consider the serial position of the bug, we observe a correlation of .79 with both programs combined and .75 and .92 for CORR and TALL, respectively ($p<.005$ for all correlations). Taken together, these observations suggest a hypothesis concerning the processes involved in debugging.

On the initial passes through a program, we will assume that a subject successfully integrates the propositions at the higher levels in the hierarchy. Successive passes lead to the integration of propositions at deeper and deeper levels. For example, on the first pass through a program, a subject may attend to propositions in levels 1, 2, and 3 and successfully integrate those at level 1; the second pass covers levels 2, 3, and 4 and level 2 propositions are successfully integrated.

Although greatly simplified, this example is intended to illustrate an important point. The probability of successfully integrating a proposition into a memory structure decreases as depth increases. By making multiple passes over propositions at lower levels, the probability of an eventual successful integration is increased. Obviously, if the proposition that represents a bug is not integrated, then the bug will not be found. The depth of a bug should correlate, therefore, with the probability of detecting the bug.

In Table 10, we have listed, for each experimental condition, the number of cases in which the bug was not found. The correlations between these numbers and bug depth are .60 and .61 for TALL and CORR, respectively. The corresponding correlations of bugs not found with serial position are .37 and .66. Recall that program TALL is the only program in which we could adequately place bugs at different depths. This analysis suggests that serial position primarily affects the time required to locate a bug while depth affects the probability of locating a bug.

The theoretical framework outlined in this paper suggests another useful measure of debugging performance. Participants occasionally made incorrect guesses, which we will refer to as errors, about the location of the bug. If, as suggested above, the time to locate a bug is a function of the number of propositions that must be processed in order to find the bug, then the times associated with these errors should be a function of the number of propositions that precede the indicated, incorrect location. That is, the serial position effect should also result in a positive correlation between the time at which an error occurs and the number of the proposition that represents the indicated location. This analysis is slightly hampered by the fact that participants were asked only to indicate the line that contained the bug and a line can contain several propositions. In this analysis, therefore, we consider the error as being associated with the first proposition in the indicated line.

Overall, there were 102 errors, 39 in TALL and 63 in CORR. The observed correlation between the times at which errors occurred and the location, or serial position, of errors, considering all errors, was .52 ($p<.005$). Considering the two programs separately, this correlation was .16 for TALL and .58 for CORR.

Participants were also asked to correct the line containing the error after they had found the bug. Our observations were similar to those reported by Gould and Drongowski (1974); subjects were able to make the appropriate corrections fairly quickly. Overall, 59% of all bugs were corrected immediately after locating the bug and 86% were corrected within 2 minutes. There were no apparent differences due to any of the factors involved in this study.

DISCUSSION

The primary motivation underlying this research was to investigate the feasibility of applying a theory of the comprehension and memory of text to a programming task. Our results indicate that such an application is both feasible and useful. We have not attempted to exploit all of the theoretical concepts that have been developed in text memory research. We used only the concepts of propositions and propositional hierarchies since these are the necessary initial aspects of a more complete theory. Additional research directed at expanding the framework discussed here would be fruitful.

The concept of a "macrostructure," discussed in a prior section of this paper, appears to be especially relevant to future research. Although the present experiment was not designed to explore macrostructures, the existence of program macrostructures is suggested by our results. There was a fairly substantial decline in debug times on the second program presented to participants as compared to debug times on the first program. Since both programs were statistical routines, their macrostructures are probably quite similar. Although some of this general improvement is probably due to increased task familiarity, practice, etc., it seems likely that some of this general improvement is also due to the fact that a macrostructure must be initially developed in order to understand the first program and this macrostructure need only be modified, rather than a new macrostructure constructed, in order to understand the second program. The reduced debug times reflect the fact that less effort is required in comprehension.

When less effort is required for comprehension, a subject can devote more effort to using problem-solving processes. This implies that there is a relation between the macrostructure, which is the result of comprehension processes, and problem-solving processes. In order to illustrate this relationship, we will consider what type of problem-solving processes could be involved in program debugging.

A very inefficient, but frequently used, problem-solving process is a generate-and-test method. With this method, a problem solver generates a potential solution and tests to determine if it is, in fact, a solution. The "safe problem" is a good example of this method. Imagine a safe having ten independent dials, each with 100 numbers. In order to open the safe, we could try each of the possible 100^{10} combinations and one of these would succeed. Although a generate-and-test method is an algorithm, in that it always yields a solution, it is rarely directly applied to such large problems.

One characteristic of human problem solvers is that they generally use heuristic, rather than algorithmic, methods. Heuristics are useful only if they reduce the number of alternatives that must be considered. Suppose, for example, that each dial of the safe were defective and a click could be heard when it was in the correct position. This reduces the number of alternatives to be considered to 10 (dials) x 100 (numbers), or 1000, possibilities. The clicks provide a valuable clue and a problem solver can use this clue to conduct a heuristic, rather than algorithmic, search.

In a debugging task, a strict algorithmic approach would require considering each element (character) of each line and testing to see if this is, in fact, the bug. This is obviously not the case. Programmers use clues, or cues, in selecting those sections of the code where a bug is most likely to be found. These cues appear to be derived from the macrostructure. Admittedly, a programmer often uses "surface structure" cues, such as output results, to detect cues. We are suggesting, however, that even such cues are related through the macrostructure rather than directly to the code.

In many cases, and debugging appears to be such a case, a problem solver does not attack a problem directly, but attempts to decompose it into relatively independent subproblems. Macrostructure propositions correspond to these subproblems. In statistical programs, for example, these macrostructure propositions and subproblems are typically "find the mean," "calculate the standard deviation," etc. The use of subproblems reduces the number of alternatives that must be considered, much as the click in the defective safe dials reduces a very large search problem to ten very much smaller search problems.

In a programming task, the use of subproblems, or macrostructure propositions, can reduce search even further. Notice that a problem solver can determine that a program is incorrect on the basis of the macrostructure alone. These propositions represent the subproblems that are required to accomplish the overall goal of the problem. If one of these subproblems is determined to be inappropriate, inadequate, or incorrect, then the actual code that must be inspected is severely reduced.

The general improvement that we observed in second program debugging, therefore, is due, in part, to increased resources that can be allocated to problem-solving processes. After debugging the first program, participants developed an appropriate macrostructure. When presented with the second program, which was similar to the first, they can determine more quickly problems with the apparent macrostructure of the second program. Debugging efforts then tend to focus on the area of code corresponding to that subproblem associated with the suspect macrostructure proposition.

In the discussion above, we have illustrated a problem that is widely recognized as a major problem area in psychology -- the interface between knowledge representation and knowledge utilization. Although we have brought this problem area into clearer focus than most previous research, a great deal of research is necessary to resolve a question of this magnitude, and we offer our comments as suggestion rather than conclusion.

Another aspect of problem-solving processes is also apparent in our results. Array bugs were the most quickly detected on both the first and second programs and showed the greatest decrease in debug times between the first and second programs. We attribute this result to a problem-solving process that we will call, for lack of a better term, a "quick shot" strategy. By definition, an array bug is one that causes an array limit to be exceeded. This is a non-syntactic bug that can be detected without actually comprehending a problem; a subject need only compare the FORTRAN DIMENSION statement with all uses of arrays. If this strategy is effective, a problem solver saves a great deal of effort in debugging. There is, apparently, a very strong tendency to initially adopt this strategy. If this strategy is successful on the first problem, there is an even greater tendency to use it on the second.

The effects of macrostructures are potentially much greater in the Gould and Drongowski study. In their study, each participant debugged each of four programs three times. The similarities between these programs facilitate the construction of appropriate macrostructures and the repeated exposure to the same program greatly facilities the comprehension process. This is reflected in the fact that Gould and Drongowski report that debug times were about twice as fast on the second presentation of a given program as on the first presentation.

Although the present experiment was designed as a partial replication of the Gould and Drongowski study, our elimination of repeated presentation of the same program makes a direct and unambiguous comparison of the results of these two studies difficult. As was stated in the introduction, it is difficult to separate knowledge representation issues and knowledge utilization issues in a given task domain. That is, program comprehension and debugging involve both the development of cognitive structures and the utilization of problem-solving processes. It seems reasonable that when a program is first encountered, a programmer will concentrate primarily on comprehending the program; that is, on developing an appropriate propositional representation. If the same, or a similar, program were presented again at a later time, the programmer could devote more effort to the problem-solving processes used in debugging.

In the present experiment, when we consider only the first program presented to participants, and where we assume program comprehension is of primary importance, we observed significant differences attributable to propositional factors but no differences due to the class of bug. On the second program, however, where problem-solving processes become more important, the class of bug was an important factor in debugging performance. Although the relative difficulty of the classes of bugs differs, this is similar to the result reported by Gould and Drongowski. In their study, participants debugged several listings, and it is probable that performance eventually became more an effect of problem-solving processes than of the development of cognitive structures to represent the programs.

It appears, therefore, that the differences due to type of bug observed in both of these studies are the result of problem-solving processes rather than of memory representation issues. Neither study, however, allows for an adequate investigation of these processes.

The fact that we observed a different order of difficulty as a function of class of bug than that reported by Gould and Drongowski requires some explanation. Recall that Gould and Drongowski considered only the class of bug while we controlled also for position in the propositional hierarchy. In both this study and the Gould and Drongowski study, array bugs were the most quickly detected; this is consistent with our discussion above of the "quick shot" strategy.

The principal difference between the results of these two studies was that Gould and Drongowski found iteration bugs to be much easier than assignment bugs while we found them to be fairly equal in difficulty. Our finding of no difference is consistent with our hypothesis. Gould and Drongowski claim that detecting assignment bugs requires a more de-

detailed comprehension of the program. This is true, however, only within the context of the specific bugs they used and probably is not true in general.

Consider first the iteration bugs introduced in their study. In one program (CORR) the bug involved a change in a DO-loop index. The resulting code was

```
DO 220 J=1, JK
JK=JK+1
   .
   .
   .
```

In a second program (TALL) a statement that initialized the elements of an array was moved to just outside the DO-loop controlling the initilization. In the other two programs, the iteration bugs were similar. These appear to be bugs that can be detected on the basis of a general understanding of the programming language used and without a detailed understanding of the program.

The assignment bugs used by Gould and Drongowski, however, are quite different. These bugs tend to require knowledge of statistical formulas in addition to understanding the program. The assignment bug in program CORR, for example, resulted in an incorrect computation of a standard deviation. The fact that these bugs require statistical knowledge in addition to program comprehension is indicated by the fact that there is not a substantial decline in debug times for assignment bugs in the condition in which Gould and Drongowski told participants the number of the line that contained the bug, compared to the condition in which no aids were given, although there were substantial declines for array and iteration bugs. Even when a programmer can allocate the majority of his resources to problem-solving processes, rather than comprehension processes, performance is not facilitated if finding a solution requires knowledge that the programmer does not possess.

The ordering of classes of bugs reported by Gould and Drongowski appears, therefore, to be due to the specific bugs used. These bugs make very different demands on a subject in terms of the amount of comprehension that is required and the problem-solving processes that are effective.

The final issue to be considered is the generality of the results reported here. One question is whether the one-page listings used here generalize to longer programs; a second concerns the fact that participants knew that there was a single bug in each listing. These are questions that cannot be satisfactorily resolved without further research.

An additional question centers on the fact that participants did not debug programs they had written. If this were the case, participants would previously have developed detailed representations of programs, and debugging would primarily be a function of problem-solving processes. This is an interesting problem, since it emphasizes problem-solving processes, or debugging skills, rather than program comprehension, but the methodological problems associated with conducting controlled research with such a task would be difficult to overcome.

Generalization from our subject population to others may also be difficult. The students involved in the present study could be characterized as "experienced" or "moderately experienced", they were not, however, "highly experienced."

Prior to conducting the study reported in this paper, we conducted a pilot study with 12 programmers who have more extensive experience and who routinely engage in program writing and debugging. This allows a comparison of the results reported here with those obtained from more experienced programmers.

Overall, the mean debug time for the study reported above was 25.2 minutes and for the more experienced programmers, the mean debug time was 23.1 minutes. Considering all experimental conditions, the correlation between the mean debug times for these two samples was .55 ($p < .05$). Although this seems to indicate that the results reported here generalize to more experienced programmers, there are obvious performance differences to be expected as a function of experience and the theoretical framework proposed in this paper could be useful in investigating these differences.

Although experience may generally be thought to correlate positively with debugging performance, we will briefly consider one case where experience appears to be negatively correlated with performance. With the high array bug in program TALL, our undergraduate sample required, on the average, 11.9 minutes to locate this bug, while our more experienced sample had a mean debug time of 43.7 minutes.

This particular bug (see Appendix A for the program listing and Table 2 for a description of the bug) causes array limits to be exceeded in an initialization procedure and the corresponding program statements are separated by comment statements from the remainder of the program and clearly identified, also by a comment statement, as an initialization procedure. Consider now how the theoretical framework proposed in this paper might be useful in explaining this result.

When asked why this particular bug was so difficult to detect, our more experienced programmers replied that this was "obviously an initialization procedure," that "initialization is trivial," or that "no one makes mistakes in initializing variables." It appears that these programmers quickly glanced at this segment of the program, immediatley recognized that it was an initialization procedure, and encoded it in memory in a form similar to "INITIALIZE RELEVANT VARIABLES." The less experienced students, who presumably use a set of propositional mappings similar to those described earlier, tend to process this segment line by line and proposition by proposition, and therefore, are more likely to discover the bug.

Thus, more experienced programmers appear to rely heavily on contextual cues. When they recognize an initialization procedure, they retrieve from long-term memory a description of what such procedures <u>must</u> do and tend to perceive the actual code as accurately performing these functions. A clear description of the role of context in processing text is provided by Lindsay and Norman (1972, pg. 133):

> "Context is the overall environment in which experiences are embedded. You know many things about the material you are reading, in addition to the actual patterns of letters on this page. You know that it is written in English, that it is discussing psychology in general and the psychology of pattern recognition in particular. Moreover, you have probably learned quite a bit about the style of writing, thus, as your eyes travel over the page, you are able to make good prediction the words you expect to see. These predictions are good enough so that you automatically fill in the missing "about" or "of" in the previous sentence, or, equivalently, not even notice its absence."

The results of our pilot study indicate that a similar effect can occur with programs.

SUMMARY AND CONCLUSIONS

In this paper, we have focused on computer program debugging and comprehension. Our emphasis has been on the representation of computer programs in memory, using a framework developed to study text memory.

Paralleling research in text memory, we have assumed that the abstract representation of programs consists of propositoins. The research reported in this paper involved experimental manipulations that were based on an assumed propositional representation. The fact that observed results were explained in terms of the properties of a propositional representation tends to support its existence. We have not, however, adequately defined what constitutes a proposition or how propositions are related in a hierarchy. Additional attention needs to be given in such questions as how experience or other individual differences affect propositions and whether the form and content of propositions for computer programs is independent of the language in which they are implemented.

Our original conjecture was that level in the propositional hierarchy was the most appropriate metric of bug location. Subsequent analyses, however, indicated that this measure is highly correlated with the number of propositions that precede the bug and also with the number of the line that contains the bug. Whether this result is a function of programs in general or only of the programs used in this study remains to be determined. The resolution of this issue appears to be an essential prerequisite of further research in this area.

The fact that the results of the first program debugged by participants strongly supports the theoretical framework proposed in this paper while the support provided by the second debugged program is not quite as strong suggests an additional area where research is needed. Future research in this area should carefully consider the construction of repeated measures designs in order to more clearly separate the effects of knowledge representation and knowledge utilization. This approach offers the advantage of making the study of program comprehension and debugging theoretically more tractable.

We feel that there are several benefits that follow quite directly from the theoretical framework underlying our research. Intuitively, the memory representation of programs is clearly related to the determination of appropriate programming language features and programming practices. In addition, comprehension of a program is an obvious prequisite to most debugging and maintenance programming tasks. A theory that relates comprehension to particular program features could allow the development of programs that are more easily debugged and more modifiable, by indicating which programming practices most aid in these tasks. Our results suggest that the location of a bug in a program's underlying propositional hierarchy is a principal factor affecting performance in a debugging task. This result has implications for the manner in which programs should be written. Specifically, programs should be written to minimize the depth to which the propositional hierarchy extends and should be modularized to reduce the number of propositions contained in a single module. Also, a theory of program comprehension based on propositional structures may prove useful in evaluating different techniques of program documentation. Documentation techniques that present information in a form closely related to its ultimate memory representation should allow faster comprehension and result in fewer comprehension errors.

In this paper, we have attempted to close the gap between basic and applied research by applying a basic theory of text comprehension to the applied task of program comprehension and debugging. Such an application, of course, requires more research than the single experiment reported in this paper. We have, however, demonstrated that theories and techniques derived from cognitive psychology can be successfully applied to software-related tasks and, hopefully, this will lead to more meaningful research in this area.

REFERENCES

Bartlett, F.C. Remembering. Cambridge, England: University Press, 1932.

Brooks, R. A model of human cognitive behavior in writing code for computer programs (2 Vols.). Pittsburg, Pennsylvania: Carnegie-Mellon University, Department of Computer Science, May 1975. (NTIS No. AD A013582)

Bulut, N., Halstead, M.H., & Bayer, R. Experimental validation of a structural property of FORTRAN algorithms (Technical Report No. CSD-TR-115). Lafayette, Indiana: Purdue University, Department of Computer Science, 1974.

Chase, W.G., & Simon, H.A. Perception in chess. Cognitive Psychology, 1973, 4, 55-81.

Gordon, R.D., & Halstead, M.H. An experiment comparing FORTRAN programming times with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.

Gould, J.D. Some psychological evidence on how people debug computer programs (Research Report RC-4542). Yorktown Heights, New York: IBM Watson Research Center, September 1973.

Gould, J.D. Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 1975, 7, 151-182.

Gould, J.D., & Drongowski, P. An exploratory study of computer program debugging. Human Factors, 1974, 16, 258-277.

Grant, E.E., & Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Transactions on Human Factors in Electronics, 1967, HFE-8, 33-48.

Kernigan, B.W., & Plauger, P.J. The elements of programming style. New York: McGraw-Hill, 1974.

Kintsch, W. The representation of meaning in memory. Hillsdale, N.J.: Erlbaum, 1974.

Kintsch, W. Comprehension and memory of text. To be published in W.K. Estes (Ed.), Handbook of learning and cognitive processes, Vol. 5, Hillsdale, N.J.: Erlbaum, 1977 (Also: (Report No. 50). Boulder, Colorado: University of Colorado, Institute for the Study of Intellectual Behavior, 1976).

Kintsch, W., & Kennan, J.M. Reading rate and retention as a function of the number of propositions in the base structure of sentences. Cognitive Psychology, 1973, 5, 257-274.

Kintsch, W., & Van Dijk, T.A. Comments on se rapelle et on resume des histories. Languages, 1975, 40, 98-116.

Lindsay, P.H., & Norman, D.A. Human information processing. New York: Academic Press, 1972.

Love, L.T., & Bowman, A.B. An independent test of the theory of software physics. SIGPLAN Notices, 1976(Nov.), 11(11), 42-49.

Mandler, J.M., & Johnson, N.S. Remembrance of things parsed: Story structure and recall. Cognitive Psychology, 1977, 9, 111-151.

Miller, L.A., & Becker, C.A. Programming in natural English (Technical Report RC-5134). Yorktown Heights, New York: IBM Watson Research Center, November 1974.

Norman, D.A. Memory, Knowledge, and the Answering of Questions. In R. L. Solso (Ed.), Contemporary Issues in Cognitive Psychology. Washington, D.C.: Winston, 1973.

Paige, J.M., & Simon, H.A. Cognitive processes in solving algebra word problems. In B. Kleinmutz (Ed.), Problem solving: Research, method, and theory, New York: Wiley, 1966.

Schneiderman, B. Expoloratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 1976, 5(2), 123-143.

Schneiderman, B. Measuring computer program quality and comprehension (Technical Report No. 16). College Park, Maryland: University of Maryland, Department of Information Systems Management, February, 1977.

Schneiderman, B., & Mayer, R. Towards a Cognitive Model of Programmer Behavior (Technical Report No. 37). Bloomington, Indiana: Indiana University, Computer Science Department, August 1975.

APPENDIX A

Program TALL

```
C
C     ...................................................................
C            PURPOSE OF PROGRAM
C                 TO CALCULATE TOTALS, MEANS, STANDARD DEVIATIONS, MINIMUMS,
C                 AND MAXIMUMS FOR EACH VARIABLE IN A SET (OR A SUBSET) OF
C                 OBSERVATIONS.
C     ...................................................................
C            INSTRUCTIONS TO SUBJECTS
C                 NO = NUMBER OF OBSERVATIONS = 30
C                 NV = NUMBER OF VARIABLES = 20
C                 ASSUME DATA FROM THE NO X NV MATRIX ARECORRECTLY READ
C                 INTO CORE IN A VECTOR A(1) - A(600).
C                 S(1) - S(30) = INPUT VECTOR INDICATING A SUBSET OF A.  ONLY
C                 OBSERVATIONS WITH A NON-ZERO S(J) ARE CONSIDERED.
C     ...................................................................
C
      DIMENSION A(600),S(30),TOTAL(20),AVER(20),SD(20),VMIN(20),VMAX(20)
C
      NV=20
      NO=30
C          CLEAR OUTPUT VECTORS AND INITIALIZE VMIN,VMAX
C
      IER=0
      DO 1 K=1,NV
      TOTAL(K)=0.0
      AVER(K)=0.0
      SD(K)=0.0
      VMIN(K)=1.0E10
    1 VMAX(K)=-1.0E10
C
C          TEST SUBSET VECTOR
C
      SCNT=0.0
      DO 7 J=1,NO
      IJ=J-NO
      IF(S(J)) 2,7,2
    2 SCNT=SCNT+1.0
C
C          CALCULATE TOTAL,MINIMA,MAXIMA
C
      DO 6 I=1,NV
      IJ=IJ+NO
      TOTAL(I)=TOTAL(I)+A(IJ)
      IF(A(IJ)-VMIN(I)) 3,4,4
    3 VMIN(I)=A(IJ)
    4 IF(A(IJ)-VMAX(I)) 6,6,5
    5 VMAX(I)=A(IJ)
    6 SD(I)=SD(I)+A(IJ)*A(IJ)
    7 CONTINUE
C
C          CALCULATE MEANS AND STANDARD DEVIATIONS
C
      IF(SCNT)8,8,9
    8 IER=1
      GO TO 15
    9 DO 10 I=1,NV
   10 AVER(I)=TOTAL(I)/SCNT
      IF(SCNT-1.0) 13,11,13
   11 IER=2
      DO 12 I=1,NV
   12 SD(I)=0.0
      GO TO 15
   13 DO 14 I=1,NV
   14 SD(I)=SQRT(ABS((SD(I)-TOTAL(I)*TOTAL(I)/SCNT)/(SCNT-1.0)))
   15 STOP
      END
```

Program CORR

```
C
C     ...............................................................
C           PURPOSE OF PROGRAM
C              TO COMPARE MEANS,STANDARD DEVIATIONS,SUMS OF CROSS-
C              PRODUCTS FOR DEVIATIONS,AND CORRELATION COEFFICIENTS OF
C              EACH VARIABLE IN A SET OF OBSERVATIONS.
C     ...............................................................
C      INSTRUCTIONS TO SUBJECTS
C              N = NUMBER OF OBSERVATIONS = 30
C              M =NUMBER OF VARIABLES = 20
C              ASSUME DATA FROM THE N X M MATRIX ARE CORRECTLY READ INTO
C              CORE IN A VECTOR X(1) - X(500).
C     ...............................................................
      DIMENSION X(500),XBAR(20),STD(20),RX(400),R(400),B(20),D(20),T(20)
C
C
C      INITIALIZATION
C
      M=20
      N=30
      DO 100 J=1,M
      B(J)=0.0
  100 T(J)=0.0
      K=(M*M+M)/2
      DO 102 I=1,K
  102 R(I)=0.0
      FN=N
      L=0
C
  105 DO 108 J=1,M
      DO 107 I=1,N
      L=L+1
  107 T(J)=T(J)+X(L)
      XBAR(J)=T(J)
  108 T(J)=T(J)/FN
C
      DO 115 I=1,N
      JK=0
      L=I-N
      DO 110 J=1,M
      L=L+N
      D(J)=X(L)-T(J)
  110 B(J)=B(J)+D(J)
      DO 115 J=1,M
      DO 115 K=1,J
      JK=JK+1
  115 R(JK)=R(JK)+D(J)*D(K)
C
C      CALCULATE MEANS
C
  205 JK=0
      DO 210 J=1,M
      XBAR(J)=XBAR(J)/FN
C
C      ADJUST SUMS OF CROSS-PRODUCTS OF DEVIATIONS
C      FROM TEMPORARY MEANS
C
      DO 210 K=1,J
      JK=JK+1
  210 R(JK)=R(JK)-B(J)*B(K)/FN
C
C      CALCULATE CORRELATION COEFFICIENTS
C
      JK=0
      DO 220 J=1,M
      JK=JK+J
  220 STD(J)=SQRT(ABS(R(JK)))
      DO 230 J=1,M
      DO 230 K=J,M
      JK=J+(K*K-K)/2
      L=M*(J-1)+K
      RX(L)=R(JK)
      L=M*(K-1)+J
      RX(L)=R(JK)
      IF(STD(J)*STD(K)) 225,222,225
  222 R(JK)=0.0
      GO TO 230
  225 R(JK)=R(JK)/(STD(J)*STD(K))
  230 CONTINUE
C
C      CALCULATE STANDARD DEVIATIONS
C
      FN=SQRT(FN-1.0)
      DO 240 J=1,M
  240 STD(J)=STD(J)/FN
C
C     COPY THE DIAGONAL OF THE MATRIX OF SUMS OF CROSS-PRODUCTS OF
C      DEVIATIONS FROM THE MEANS.
C
      L=-M
      DO 250 I=1,M
      L=L+M+1
  250 B(I)=R(L)
      STOP
      END
```

APPENDIX B

INSTRUCTIONS TO SUBJECTS

Procedure:

Tell Subjects:

1. This is not a test of your intelligence. We are interested only in observing how experienced programmers debug programs. All information regarding your performance will be treated as confidential.

2. You will be given two program listings, one at a time.

3. Each listing is syntactically correct.

4. Each listing contains a single error in one line.

5. The I/O statements have been removed but you may assume that the data are stored in the computer correctly.

6. Try to find the line containing the bug as QUICKLY and ACCURATELY as possible.

7. As soon as you think you have found the line containing the bug, show me which line it is.

8. I will then tell you whether or not you are correct.

9. After you have finished the first program, I will give you the second program.

10. If you like, you may take a rest break between these programs. However, once you begin a program you must continue to work on it until you find the bug.